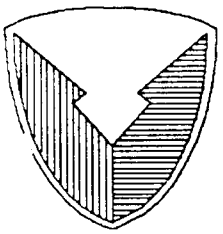


CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: - **REUSE TOOLS TO SUPPORT ADA INSTANTIATION CONSTRUCTION**

Final Report

CIN: C02087KV000100

1 JUNE 1990

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1 Jun 90	Final Report

4. TITLE AND SUBTITLE

Reuse Tools to Support Ada Instantiation Construction

5. FUNDING NUMBERS

DAAL03-86-D-0001

6. AUTHOR(S)

Patrick J. Merlet

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Productivity Solutions, Inc.
122 N. 4th Ave.
Indialantic, FL 32903

8. PERFORMING ORGANIZATION REPORT NUMBER

1293

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

US Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

89037

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The primary objective of this effort was to investigate the feasibility of developing a generalized component comstruction capability, building on the concepts and experiences of the CAMP effort. This report provides a detailed analysis of the CAMP project and component constructors. A broad investigation of alternative approaches is presented, along with a detailed investigation of reuse environment integration. Software adaptation is identified as an essential aspect of reuse. A detailed approach is proposed for the development of a domain-specific adaptation and reuse capability, with specific recommendations for future R&D.

14. SUBJECT TERMS

Software Adaptation and Reuse, Programming Language Evolution, CAMP, Component Construction, Ada, Composition

15. NUMBER OF PAGES

100

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to **stay within the lines to meet optical scanning requirements.**

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE - See authorities.
NASA - See Handbook NHB 2200.2.
NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - DOD - Leave blank.
DOE - DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA - NASA - Leave blank.
NTIS - NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

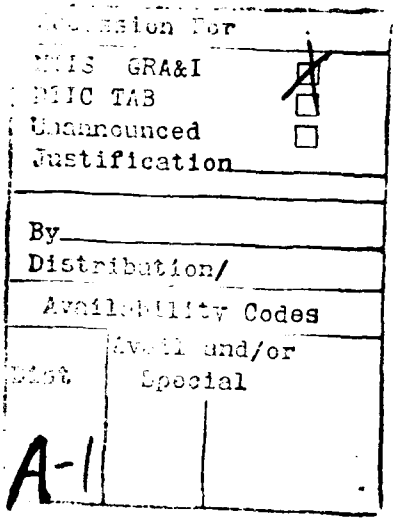
Reuse Tools to Support Ada Instantiation Construction

Final Technical Report

Prepared by

Patrick J. Merlet

Software Productivity Solutions, Inc.
122 4th Avenue
Indialantic, FL 32903



Prepared for

U.S. Army Communications-Electronics Command (CECOM)
Center for Software Engineering
ATTN: AMSEL-RD-SE-AST-SS
Ft. Monmouth, NJ 07703-5204

June 1, 1990

Contract No. DAAL03-86-D-0001
Delivery Order 1293
Scientific Services Program

Executive Summary

The purpose of this research project is to investigate the feasibility of developing *Reuse Tools to Support Ada Instantiation Construction*, building on the experiences of the Common Ada Missile Packages (CAMP) program. The work has been performed by Software Productivity Solutions, Inc., of Melbourne, Florida for the U.S. Army Communications-Electronics Command (CECOM).

While meeting the objectives of this project, we did not strictly adhere to the four tasks outlined in our approach to meet these objectives. Our primary objective was to investigate the feasibility of developing a *generalized* component construction capability that relieves some of the problems of domain-dependence and part maintenance, building on the concepts and experiences of the CAMP effort. Our secondary objective was to investigate the integration of component construction technology with existing and emerging software development environments.

The four tasks (see Section 1.2) were specified as follows: 1) analyze the CAMP methodology, 2) define the requirements for a generalized constructor capability, 3) investigate development environment integration, and 4) demonstrate the feasibility of the generalized constructor capability. Towards the end of Task 1, we began to investigate alternative generalized constructor approaches, in effect opening a Pandora's box. Although thought to be a relatively straightforward exercise, we were faced with the realization that selecting an appropriate alternative approach was infeasible.

There are several reasons why we were unable to select an alternative generalized approach, all of which stem from unexpected research findings. First, we learned that the CAMP constructors were hard-coded. This was a real surprise. It was generally assumed that there was a knowledge-based component schema representation and rule-based component construction foundation upon which we would be able to refine and build upon. Secondly, it was assumed that the selected approach would be a matter of choosing an appropriate constructor-constructor type strategy to solve the maintainability problem, and incorporating and encapsulating domain knowledge where appropriate to relieve the domain-dependence problem. Once we began to explore alternative approaches outside of the CAMP arena, we gained a new perspective of constructors. We learned of the central importance of adaptation (and how little we know about it), the implications of programming languages and their evolution, and the necessity of a domain analysis for the development of a domain-specific adaptation and reuse capability. As a result of all of this, it was not possible to proceed with

Task 2. However, our newfound insight provided us with a better understanding of constructors and their generalization.

We then proceeded to Task 3, the investigation of development environment integration. The expert system investigation led us to the conclusion that we have yet to develop a sufficient understanding of the expertise necessary to adapt reusable software components for specific applications. If, after developing the GCC requirements and/or a GCC specification, it becomes clear that some aspect(s) of the GCC are best suited for an expert system, then it would be appropriate to impose expert system implementation requirement(s).

The reuse environment investigation turned out to be surprisingly interesting and educational. Our previous work (from Task 1) had taught us of the synergy between the GCC, adaptation, and reuse. During Task 3, we built upon this observation, learning that not only was there a good integration between the GCC and reuse environments, but that reuse environments have a much needed automated adaptation capability. This investigation grew to include a detailed operational scenario and a significant amount of analysis. Thus, while we have not addressed Task 4 in the manner originally planned, we believe that this effort (Task 3) serves as a demonstration of the GCC's feasibility.

As far as defining the GCC requirements (as called for in Task 2), we believe that the scope of the requirements definition was larger than anticipated, and additional research is required. We have observed that there is an enormous amount of basic research in progress which is potentially applicable to the automated adaptation and reuse of software components. While promising, none yet have demonstrated the capability to scale-up to the demands of software engineering in-the-large.

We strongly recommend that CECOM continue to fund this research and development effort. We consider the current effort to have been a necessary but preliminary effort. Domain analysis has been identified as essential to the development of a domain-specific adaptation and reuse capability, while adaptation has been identified as essential for software reuse in general. In addition, continued R&D of specific programming languages/systems holds great potential for achieving advanced software adaptation and reuse.

A new strategy is proposed for developing the GCC. The new approach includes a significant amount of research and detailed study, including the following three major thrusts: a C^2 domain study; exploratory software adaptation research; and a detailed study of programming languages/systems. We believe this level of effort is necessary to successfully derive the GCC requirements.

Table of Contents

List of Figures

List of Tables

1 Introduction

The purpose of this report is to investigate *Reuse Tools to Support Ada Instantiation Construction*, and the feasibility of developing a *generalized* component construction capability, building on the concepts and experiences of previous research. The following sections identify the problem, describe our objectives and approach, and specify the format of the report.

1.1 Problem Statement

The Common Ada Missile Packages (CAMP) project (primarily funded by the STARS Joint Program Office, sponsored by the Air Force Armament Laboratory, and performed by McDonnell Douglas) produced methods, tools, and software parts to support the construction of missile systems using reusable software technology. A primary result of this effort was the development of a prototype automated parts composition system, called the Ada Missile Parts Engineering Expert (AMPEE) system.

According to the CAMP developers, the AMPEE "alleviates many of the problems associated with software reuse by providing the user with an expert assistant to advise him on the availability and relevance of CAMP reusable Ada software parts to his application, and to aid in the development of software systems by automatically generating the required code for particular operations or subsystems of the application." While the CAMP developers admit that "much of the AMPEE system is CAMP-specific, the underlying principles are applicable to a variety of domains." [MCN88a]

It would be advantageous to the Army to acquire the AMPEE capabilities developed by CAMP and apply them (if feasible) to its domains of interest, such as command and control (C^2). Figure 1–1 [MCN86b] provides an overview of the AMPEE system.

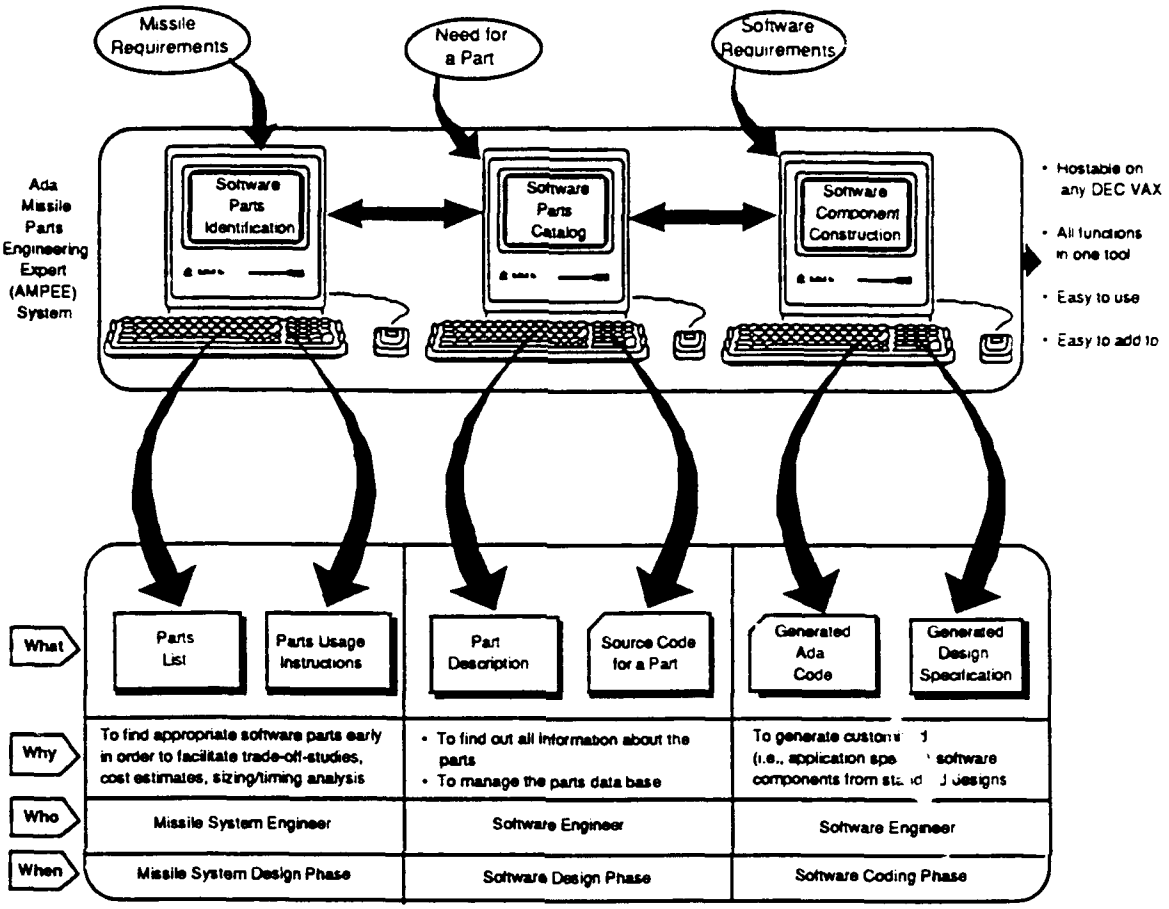


Figure 1–1 Overview of the AMPEE System

The AMPEE is composed of the following three major subsystems [MCD85a]:

- **Parts Catalog Subsystem** — provides the user with a means of entering, modifying, deleting, and disseminating information about reusable software parts
- **Parts Identification Subsystem** — assists the user in the early identification of software parts that may be relevant to his particular application; provides the capability of mapping high level user requirements to software parts
- **Component Construction Subsystem** — provides the user with a means of generating customized software components from tailorable software parts that are available in the parts catalog; also provides for the regeneration (i.e., generation based on modified tailoring requirements) of software components

Out of these three subsystems, the Component Construction subsystem shows the most promise for being "extended to another domain." [SOW] In fact, portions of this subsystem were intentionally designed to be domain-independent, and may be directly applicable to the development of Army software in other domains. However, many of the CAMP constructors are highly domain-dependent. Besides being highly domain-dependent, maintenance is also a potential problem for the CAMP constructors because they are intimately tied to the software parts they support. [MCN88a]

1.2 Objectives and Approach

The primary objective of this effort is to investigate the feasibility of developing a *generalized* component construction capability that relieves some of the problems of domain-dependence and part maintenance, building on the concepts and experiences of the CAMP effort. Another objective of this effort is to investigate the integration of component construction technology with existing and emerging software development environments. The following four tasks outline our approach to meeting these objectives.

1) **Analyze the CAMP methodology.** Of particular concern are the effort required to move the CAMP constructors to another domain, and the potential constructor maintenance problem which the CAMP developers have identified. Recommendations will be made regarding the use, potential modification, or non-use of the CAMP parts composition method and component constructors. Alternate software composition and component construction methods will be investigated, postulated, and compared to the CAMP approach. Based on this analysis, methods appropriate for the C^2 domain will be selected for further investigation.

2) **Define the requirements for a generalized constructor capability.** Based on the methods selected in Task 1, the requirements for a *Generalized Constructor Capability* (GCC) will be defined. Operational, information, processing, and interface requirements will be addressed. Quality goals and design constraints will also be identified.

3) **Investigate development environment integration.** Approaches and implications of integrating the GCC within software development environments will be investigated. In particular, the suitability of commercial off-the-shelf (COTS) software to support the GCC implementation will be determined. This investigation will be focused on expert system development environments and reuse environments.

4) **Demonstrate the feasibility of the generalized constructor capability.** The feasibility of the GCC will be demonstrated by first developing a

prototype design for the GCC, and then providing an appropriate proof-of-concept. The proof-of-concept may consist of developing and analyzing operational scenarios which illustrate the application of the GCC within a given C^2 environment.

During the course of this STAS, we were forced to modify our approach to meeting the primary objectives. Specifically, Task 2 has been deferred until additional research can be conducted. The format of the report (detailed in the following section) has been modified to reflect the actual approach which was taken in meeting the primary objectives. (For additional information, see Section 5.2.)

1.3 Format of the Report

Section 2, Common Ada Missile Packages (CAMP) Analysis, contains a detailed analysis of the CAMP project (Phase 1 and 2). Particular emphasis is placed on the CAMP constructors. A brief and preliminary analysis of the Phase 3 effort (still underway) is also provided.

Section 3, Generalized Construction Approaches, contains an assessment of software composition and construction technology, including a review of several research and development projects. This is followed by an investigation of the composition and construction of Ada components in particular. The section concludes with an historical account of the evolution of programming languages, and how this is related to component constructors.

Section 4, Development Environment Integration, documents our investigation of the integration of the GCC with both expert system development environments and reuse environments. The reuse environment investigation includes a detailed operational scenario which demonstrates how the GCC will impact future reuse library systems.

Finally, Section 5, Conclusions and Recommendations, summarizes the conclusions reached during this effort, the research effort itself, and then presents our recommendations for future research and development.

2 Common Ada Missile Packages (CAMP) Analysis

This section of the report documents our analysis of the CAMP methodology, with emphasis placed on the CAMP component construction approach. After providing the background of the CAMP effort, we then analyze the CAMP Phase 1 and 2 projects in detail, followed by a more brief analysis of the latest developments from the CAMP Phase 3 project.

2.1 CAMP Background

CAMP identified three kinds of parts that could be used to implement an application system: *simple parts*, *generic parts*, and *schematic parts*. A **simple part** is an Ada software unit which can be reused "as is." A **generic part** is an Ada software unit which allows tailoring through the use of Ada's generic facilities. Kinds of tailoring permitted by Ada generics includes: importing application-specific data types, data objects, and subprograms. A **schematic part** is a template with a set of rules for generating Ada software units. Schematic parts handle types of tailoring not supported by Ada generics. The rules for program generation are generally very application-specific, and the generated code will "custom fit" those specific needs.

Simple and generic parts are intended for use during detailed design and coding. Schematic parts represent design-level information, and are intended for use during the requirements and design phases. [MCN86b] A key feature of CAMP parts is their open architecture. The user has full visibility into underlying or contextual parts defining a given part's environment. Pre-defined and/or application-specific objects, types, operations, and packages can be selected by the user to define a part or its environment. [MCN88b]

CAMP developed two types of tools (called constructors) to support the tailoring of software parts: *generic instantiator* and *schematic parts constructor*. A **generic instantiator** assists the part user in instantiating Ada generic parts, and a **schematic parts constructor** assists the part user in generating Ada code from schematic parts. Using questions specified by the part designer, both types of constructors have a dialogue with the reuser to collect necessary application-specific inputs and create the appropriate Ada code. [MCN86b] Whether the part is a generic or schematic part is transparent to the reuser. The parts constructors can be used for "what if" analysis as well as code generation. [MCN88a]

A custom constructor is developed for each and every schematic part, and for those generic parts deemed "sufficiently complex." Many of the CAMP generic parts had complex interfaces. Complexity can be hidden from the reuser through the use of defaults, but the defaults may need to be overridden. [MCN86b] The complexity metric used to determine if a

constructor was needed was the number of generic parameters of the part. Parts with a small number of parameters are called "simple generics," and are instantiated without the use of a constructor. [MCN88a]

The constructors are part of a system called the Automated Missile Parts Engineering Expert (AMPEE) system. Despite the fact the code is generated by the AMPEE system, it is very important to note that the AMPEE primarily supports a *parts composition* approach to software reuse (as opposed to a *program generation* approach). A parts composition approach builds new application systems using actual code components, while a program generation approach generates the code for the new application system entirely from scratch.

The CAMP developers believe that universal code generation technology is not yet feasible for real-time embedded applications. In order to meet the efficiency requirements of the CAMP parts, the AMPEE constructor capabilities are very part-specific. Although the parts and constructors are domain-specific, "the top-level design of all of the constructors follows the basic paradigm of inputs-processing-outputs." [MCN88a]

Based on various CAMP documentation, we have extracted constructor-independent requirements for the CAMP part constructors. They are intended to give a feel for, and provide an understanding of, what an AMPEE constructor does. The requirements are organized as follows:

- External Interfaces, describing the external software required to support the constructor.
- User Interface, describing scenarios of system-level and constructor-specific user interaction to perform a construction task.
- Internal Data, describing the information stored internally and manipulated locally by each constructor.
- Constructor Processing, describing the processing performed by every constructor, divided into analysis and code generation phases.
- Outputs, describing the outputs of a constructor.
- Development Methodology, describing the method for developing a part constructor.
- Use of Expert System Technology, describing the use of expert system technology in implementing AMPEE and the constructors.

These topics are addressed in more detail in the following sections.

2.1.1 External Interfaces

The AMPEE requires the use of an expert system shell and a higher order language (HOL) to implement the constructor and support its run-time execution.

Interfaces to the host file system are used to access fixed code fragments used in part construction, and to write the generated part code to source files. This capability is handled using the input/output facilities of the HOL.

Interfaces to a parts catalog subsystem are used to acquire the locations of "fixed" portions of part code. The catalog schema also contains other part information used by the constructor.

Other interfaces to host system are used to acquire user identifications and time stamps, both of which are used to tag constructor schema information.

The AMPEE also interfaces with the screen management facilities of the host system. Most of its user interface capabilities are provided through the expert system shell, although a few are accomplished via an HOL interface direct to the host system.

Finally, the AMPEE includes interfaces to an Ada compilation system for compiling the parts.

2.1.2 User Interface

The user interface to the AMPEE can be divided into system-level interaction capabilities, and interaction with constructors. The system-level interaction follows the following scenario:

- Log the user onto the system. (The system verifies validity of the user requesting services.)
- Solicit identity of the part to be constructed. (The system verifies that a constructor for the part exists.)
- Obtain the name of the file where the constructed Ada part code is to be written.
- Transfer control to the specific parts constructor and return after generation is completed.

Interaction with individual constructors follows the next scenario:

- Prompt the user for required inputs. (The constructor formats the inputs and checks conformance to input constraints. For example, some inputs may be required to be valid Ada identifiers). CAMP

used both menu-based and forms-based mechanisms to collect the input. Requests are made using a domain-oriented language.

- The user must have a knowledge of Ada to provide appropriate information for generating certain code, and must have a knowledge of the application domain to produce meaningful outputs.
- For generic instantiations, the user may have the option of selecting among pre-defined objects, types, operations and packages from the parts base, or of providing user-defined objects, types, operations and packages.

2.1.3 Internal Data

The specific user requirements are captured and stored in schema (called a "response schema" in [MCN86b] and "requirements sets" in [MCN88a]). These schemas are permanently stored for future browsing, updates, and regeneration of code.

Various intermediate data structures are used in constructing the component. Local facts are used to control the firing of rules. For generic parts, knowledge is encoded which supports part instantiation. For schematic parts, knowledge is encoded about the component blueprint, Ada coding procedures, and efficiency issues.

2.1.4 Constructor Processing

The processing performed by a constructor can be divided into two phases: an analysis phase, and a synthesis phase.

During the analysis phase, the AMPEE collects part-specific data and options, and analyzes the user inputs (part requirements) for completeness and consistency. The user inputs (part requirements) are converted into an intermediate form for further processing.

During the synthesis phase, the AMPEE generates Ada code according to the part template and user requirements. "In general, the data type definitions are generated first, followed by instantiations of CAMP parts and/or production of new code.... [C]omplexity varies considerably among constructors." [MCN88a]

Capabilities are also provided with each constructor to browse and modify requirements previously specified for a part, and to "regenerate" a component using the new requirements.

2.1.5 Outputs

Outputs from each AMPEE constructor consists of the code implementing the part, and a header for that code. The contents of a header are custom-generated for each kind of part. For generic parts, the generated code does not just include the instantiation of the generic, but also includes code providing context for the generic.

2.1.6 Development Methodology

The following methodology is defined for developing a constructor using the CAMP approach [MCN88a].

Prior to developing a constructor, there must be "an intensive analysis ... to determine if there is sufficient demand for such a constructor to warrant the non-trivial development cost. For example, the Kalman Filter Constructor comprises some 8000+ lines of Lisp/ART code and has access to another 2700 lines of code in common utilities."

Once the decision is made to implement a constructor, a constructor developer works with a part developer to define their respective requirements. Two kinds of graphical representations are developed in support of the requirements definition of the constructors: *screen flow* diagrams and *constructor high-level view* diagrams.

Screen flow diagrams are used to define a user dialogue. Their definition can help "point out omissions in the requirements and misunderstandings between the intent of the Ada part designer and the constructor designer." Note, while there are other examples of screen flow diagrams, there are no *constructor* screen flow diagrams within the CAMP documentation.

Constructor high-level view diagrams depict the CAMP parts that will be used, packages to be provided by the reuser, and packages that will be output by the constructor. It also shows the major options available to the reuser, or decisions the reuser must make. Figure 2–1 provides the only constructor high-level view diagram we found within the CAMP documentation. [MCN88a] The diagram shows that the reuser has choices to make regarding the provision of data types and operations, and must select between alternative CAMP Kalman filter parts. The output of the constructor consists of a data types package and the actual Kalman filter component.

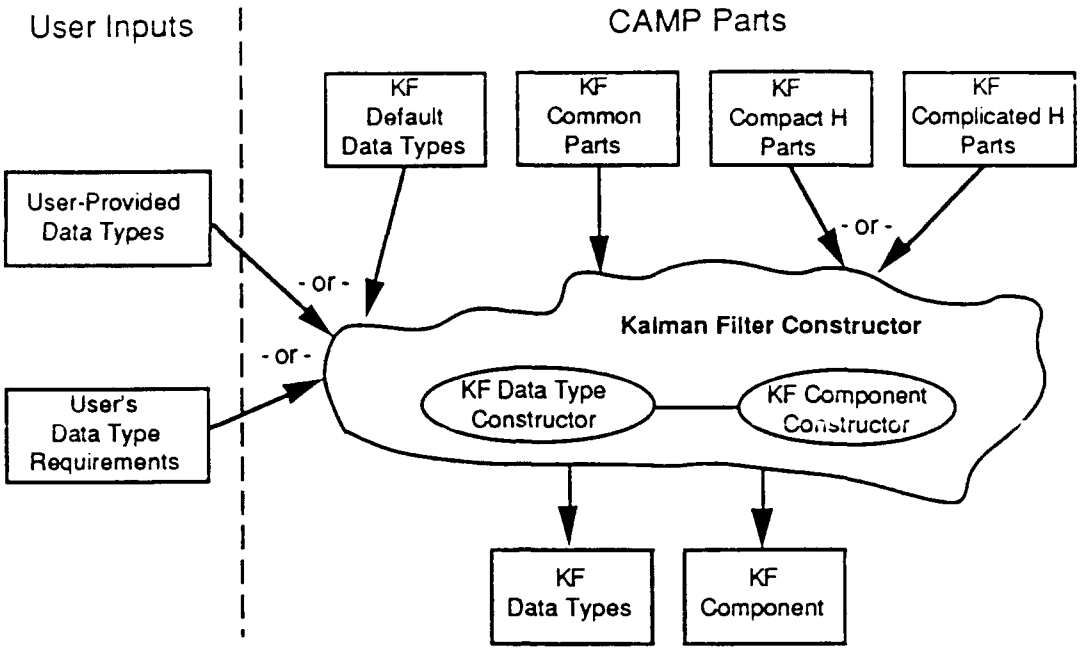


Figure 2–1 Kalman Filter Constructor — High-Level View

2.1.7 Use of Expert System Technology

A key aspect of the Phase 1 CAMP automation research was in the applicability of expert systems technology, where the researchers concluded that "[expert] systems have a high potential in the automation of the software parts engineering process ... and would be the best vehicle for building the schematic part constructors."

To be more specific, CAMP Phase 1 proposed the use of an *expert system shell* to build the AMPEE. An expert system shell provides mechanisms for expressing and storing *domain knowledge,* and an *inferencing* mechanism to process that knowledge. Domain knowledge is usually expressed in terms of *facts* and *rules*. The result is a domain-specific expert system, of which the AMPEE is an example. In this way, the CAMP developers hoped to focus on techniques for construction, and not for building expert systems. Figure 2–2 [MCN86b] illustrates the CAMP approach for using an expert system for component construction.

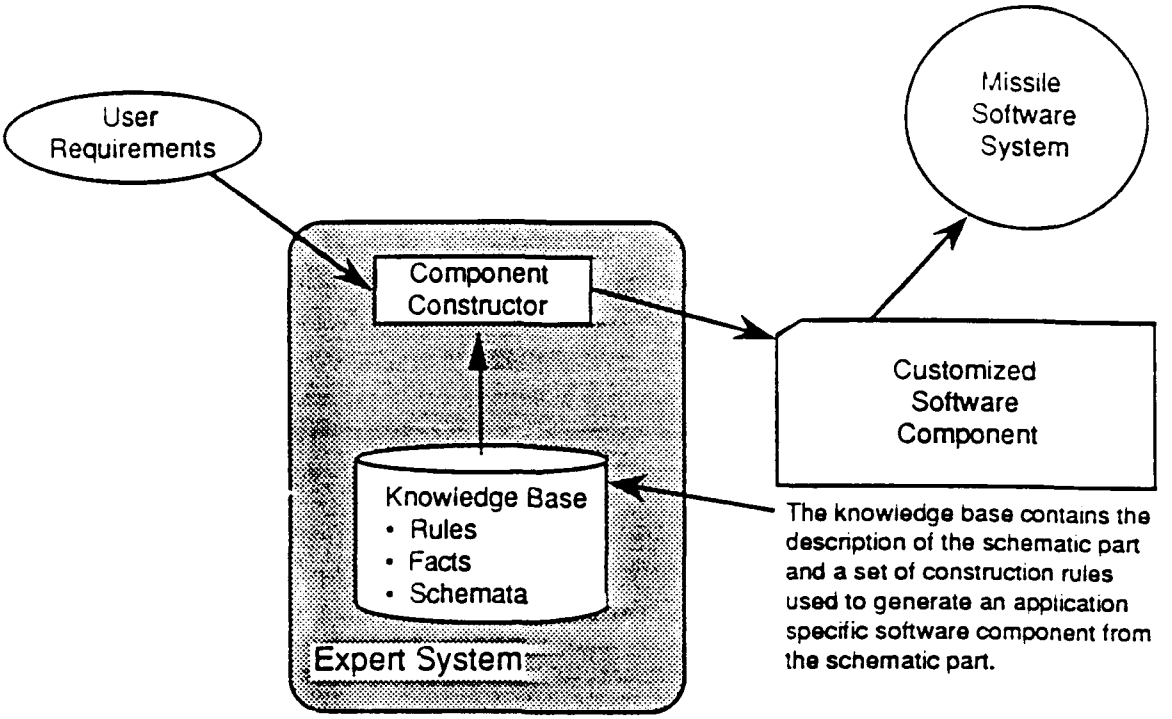


Figure 2–2 Overview of a Schematic Part Constructor

The use of an expert system over conventional software design approaches was preferred by CAMP for the following reasons [MCN86b]:

- They are better suited than conventional systems when the processes being implemented are evolutionary in nature. An expert system need only change the knowledge base when the process changes. With a conventional approach, the code must be modified.
- They are very powerful symbolic processors, requiring only a small number of rules to implement a schematic constructor.
- Most expert systems have powerful facilities for building user interfaces.

CAMP Phase 1 concluded that there are lots of capabilities where expert systems technology could be specifically applied to the parts construction system. "Expert features" that can be provided by constructors include [MCN86b]:

- Providing the expected format of input data to reusers.
- Optimizing user provided inputs (an example constructor-specific optimization is elimination of redundant state transitions from the Finite State Machine (FSM) constructor inputs).

- Checking for input correctness (an example constructor-specific check is testing for non-determinism in state transitions from the FSM inputs).
- Selecting target code constructs (for example, choice of using the "case" versus "if-then" statements).

CAMP selected the Automated Reasoning Tool (ART) and Common Lisp to support the AMPEE development and operation. The Phase 1 effort was hosted on a microVAX since VAX equipment is widely available, but various problems led to a rehosting to a Symbolics 3620 for Phase 2 (in particular, the poor performance of VAX Lisp, and a rehost of VAX ART to the C language). [MCN88a] ART was originally selected as the expert system for the AMPEE because it is a commercial product and is available on the VAX.

After Phase 2, the CAMP researchers concluded that "for the most part an expert system was not required." [MCN88a] As they developed the CAMP parts and the AMPEE system, they found sequential solutions for problems originally thought to be non-deterministic.

Actual usage of the ART expert system shell was for "data structuring (via the ART schema system), ... consistency checking and interface control (via a small number of simple forward-chaining rules), and for display of the missile software hierarchy." Many features of ART were never used by CAMP. [MCN88a]

Among the disadvantages of using an expert system shell cited by CAMP (in particular, the ART) are:

- Limited portability (ART must be available on the user's processor)
- Prohibitive cost (ART is a high-end expert system)
- Achieving and maintaining compatibility with the current version of host operating systems
- Poor startup performance (the AMPEE must be reloaded and its state reconstructed on every use of the system)
- Poor response time performance (the ART can leave the user hanging in the middle of an operation)

Early on, the CAMP developers recognized that a traditional database management system could be used for the parts library, but in their desire to share information among the AMPEE subsystems, they decided to implement this capability using ART as well. [MCN86b] However, the degree of interconnectivity of the CAMP subsystems turned out to be less than desired. [MCN88a]

2.2 Analysis of the CAMP Approach

This section provides our analysis of the CAMP parts composition and component construction approach. We begin by reviewing the conclusions and recommendations provided by the CAMP developers. We then perform a detailed analysis of the CAMP parts, parts composition, and component constructors. Next, we explore the implications of modifying the CAMP constructors to support other domains. Finally, we look at the kinds of adaptations provided by CAMP.

2.2.1 Conclusions and Recommendations of the CAMP Developers

Having conducted a multi-year research program to demonstrate the feasibility and value of reusable Ada software parts in real-time embedded (RTE) applications, the CAMP developers carefully documented their "lessons learned" in the CAMP Phase 2 Final Technical Report (FTR). Conclusions and recommendations are given concerning the Ada programming language, Ada compilers, Ada parts and part catalogs, and component constructors. Within this wealth of information, the CAMP developers provide a self-critique of several aspects of their research, suggest alternate approaches, and postulate directions for future investigation.

This material is presented here as a prelude to our analysis of the CAMP approach. Except where otherwise noted, the source of all the material in the following subsections is the CAMP-2 FTR. [MCN88a, Section VIII]

2.2.1.1 Concerning the Ada Programming Language

With respect to reusability, RTE applications, productivity, and programming-in-the-large, the CAMP developers reached the following conclusions concerning the appropriateness, effectiveness, and inherent efficiency of the Ada programming language:

- With a few minor exceptions (see the recommendations below), Ada achieves its reusability design goal.
- Ada is an effective language for real-time embedded applications.
- There appears to be no Ada features which are inherently inefficient, however, there are Ada features (e.g., Ada generic units) which require a global optimizer to be sufficiently efficient for severely constrained RTE applications.

- The use of Ada results in improved development productivity:

> [T]he productivity experienced during CAMP parts development was approximately 61% greater than that predicted by COCOMO [a software cost estimating model] for embedded software development. [MCN88a; Section II]

- Ada's support for programming-in-the-large is one of its chief advantages from a management perspective.

From the perspective of reusability, the CAMP developers recommend the following changes to the definition of the Ada language:

- Allow address objects to be passed as generic parameters.
- Allow representation clauses to be defined within a package body.
- Allow a single, unmodified Ada specification to be used with multiple bodies within a single application.
- Require a compiler to support separate compilation of generic units and subunits.
- Allow procedural data types.

2.2.1.2 Concerning Ada Compilers

With respect to reusability and RTE applications, the CAMP developers reached the following conclusions concerning the effectiveness and efficiency of Ada compilers:

- A full implementation of the Chapter 13 features of Ada is essential in RTE applications.
- Ada compilers do exist which are effective for RTE applications (e.g., the 1750A Ada compiler used on the CAMP 11th Missile Application).
- Ada compilers do exist which are effective for applications that want to use reusable software components, i.e., handle Ada generic units effectively (e.g., DEC VAX Ada compiler).
- CAMP data indicates that the current generation of Ada/1750A compilers do not support generic units well and this lack of support will hinder RTE applications that want to use reusable software components.
- CAMP data indicates that current implementations of Ada **tasking**, **generics**, and **exceptions**, are sufficiently inefficient to cause concern in severely constrained RTE applications.

- With the exception of the inefficiencies due to generic units, tasking, and exception handling, current Ada compilers appear to have efficiency equivalent to other HOL compilers used in RTE applications.
- The ability of Ada compilers to perform global optimizations is critical to the successful use of Ada and the reuse of Ada parts in RTE applications.

The CAMP developers recommend the following enhancements and precautions concerning the maturity and efficiency of Ada compilers:

- The DoD needs to enhance its Ada validation process (too many validated Ada compilers have detectable errors); during the next few years, DoD mission-critical RTE Ada projects should establish a contractual relationship with their compiler developer to reduce risk.
- The Ada validation suite must be changed to incorporate tougher tests on generic units (e.g., such as those within the CAMP benchmarks [see MCN88c]).
- Ada compilers should be able to alternate between single body and multiple body generic implementation based on either implicit or explicit information.

2.2.1.3 Concerning Ada Parts and Part Catalogs

The CAMP developers—having developed 454 parts consisting of over 16,000 lines of operational code and another 27,000 lines of Ada test code—reached the following conclusions concerning the use of Ada in the development of reusable parts:

- The use of strongly typed software parts has significant benefits to the parts user, but complicates the development of parts:

> One of the primary decisions the CAMP team had to make very early in the development of the CAMP parts was how extensively to use data typing. The chief advantage of making the parts strongly typed was the high degree of protection against misuse of the parts such typing would provide. The disadvantage of using strong typing was the increased complexity of developing the parts. The interactions between types and generics are much more complex than they appear to a casual user of Ada.
>
> Initially, we had some doubts about the use of strong typing. Was it worth the extra effort to avoid data typing errors? We surveyed some of our on-going missile projects and asked them if data typing errors were a problem. Somewhat to our surprise, we found that the misuse of data was considered to be a significant problem area. Given the large number of different types of data used in missile applications, programmers sometimes made "stupid" mistakes (e.g., mixing radians and degrees) and

these types of errors were frequently not detected until the software was tested; at this point they were very difficult to isolate. Based on this information, we decided to use strong typing in the development of the CAMP parts. After all, the parts would be developed once, but used many times.

- It costs more (5–10%) to develop reusable parts than to develop customized software.
- Software parts for RTE applications must be developed to be semi-abstract.
- The use of Ada software parts can increase productivity (by up to 15% [see MCN88b, Section III]).
- Cataloged Ada parts should be classified by logical operations, not physical Ada units:

The CAMP parts catalog was implemented so that the basic units being cataloged were Ada units. Upon reflection, and after having used this catalog, we believe this approach has two significant disadvantages.

- When viewing parts, the user gets entire Ada units and then has to locate the portions of interest; this is less than optimal.
- Too many entities are cataloged under the current scheme. This can lead to user frustration and result in the parts not being used.

We believe that a better approach would have been to catalog the logical parts, not the physical Ada code units. For example, the catalog should tell the user that it has an entry for a unbounded LIFO queue, not that it has a package specification called LIFO_QUE and a package body with the same name. Using this paradigm, the user would search for logical parts and then, if needed, the user could examine the Ada structure of these parts.

- The taxonomy(ies) used by an Ada parts catalog should be soft-coded.

The CAMP developers recommend the following organizational method of developing reusable parts:

- Parts should be developed by a parts development team driven by project needs.

2.2.1.4 Concerning Component Constructors

The CAMP developers cited the following conclusions and recommendations concerning the cost-effectiveness of capturing schematic commonality:

- Some important types of commonality cannot be captured in Ada:

> Early in the CAMP program, we realized that there were types of commonality that existed within most domains that either could not be captured using Ada alone, or could not be captured efficiently using Ada alone. We refer to this type of commonality as *schematic commonality*. To capture this type of commonality requires a tool which can build Ada code when given the requirements of a particular application. We refer to these tools as *schematic component constructors*...

- Schematic component constructors have high value:

> Based on the number of lines of code generated by the Kalman Filter Constructor for the 11th Missile Application, we estimate that a 28% productivity improvement could be obtained just from using the Kalman Filter Constructor.

- More research needs to be performed to develop an approach for building schematic component constructors:

> Although we believe that the utility of schematic component constructors is high, the current approach to their construction requires a large development effort and the resulting tool is not easily modified. [MCN88a, Section VIII]
>
> The approach used in the AMPEE system ties the constructors (for complex generic parts) intimately to parts that they utilize. This can be a problem if the part(s) on which the constructor is based change in areas that are relevant to the production of code by the constructor.
>
> An alternative that bears further exploration is the concept of a *constructor constructor*, i.e., a generalized software constructor that would generate specific constructors. One way to do this would be by embedding commands within the reusable parts themselves that would indicate the information that would be required from the user in order to generate the tailored Ada components that are needed. The parts could then be run through a preprocessor to produce the appropriate user queries. Code generators and facilities to permit data type definition or provision outside of the constructor would also be required. In essence, this would be a smarter constructor, where less of the information is hard-coded in the constructor itself. [MCN88a, Section IV]

This is somewhat similar to the approach used by the Development Arts for Real-Time Software (DARTS) program generation system, developed by General Dynamics. [MCF85] In DARTS, software is genericized by embedding domain language statements in existing source code. These statements are used to direct software generation by referencing system knowledge bases. The user enters system specifications in some domain language, and through

a series of transformations, the specifications are translated into customized application code. To support all of this, a domain language and translator must be developed, as well as the requisite knowledge bases. [MCN86b]

2.2.2 Analysis of the CAMP Parts

The CAMP developers identified three types of parts: *simple, generic,* and *schematic*. In addition, the term *meta-part* is used in association with the component constructors. A **meta-part** is defined as either a complex Ada generic or a schematic part. Each constructor in the AMPEE system is said to be based on a CAMP meta-part, and meta-parts are said to be reusable, and exist in the Parts Catalog. [MCN88a]

At best, these statements are confusing. Our investigation leads us to believe that within the Parts Catalog are non-generic parts, simple generic parts, and complex generic parts, but neither meta-parts nor schematic parts. These terms will be discussed in more detail within the CAMP constructor analysis of the following section. The remainder of this section provides an analysis of the CAMP parts and Parts Catalog, and the CAMP parts composition approach.

The following criteria were used to identify the Ada parts which were developed during the CAMP program [MCN88a]:

1) A part is a package, subprogram, or task.
2) A part must be usable in a stand-alone fashion.
3) Organizational packages are not parts; and package bodies are never parts.

Furthermore, a CAMP part may be a Top Level Computer Software Component (TLCSC), a Lower Level Computer Software Component (LLCSC), or a unit. A **TLCSC** is an outer level package or procedure — one that is not nested within another package. An **LLCSC** is a package that is nested within some other entity, generally another package. **Units** are nested procedures, functions, or tasks.

These criteria have led to a Parts Catalog which contains many (non-part) organizational packages. As a result, many parts are dependent (both explicitly and implicitly) upon other parts and non-parts. This organization is clearly less than desirable. Table 2–1 provides an example of the structure of the CAMP parts and packages within the Parts Catalog.

Table 2-1 Example CAMP Parts/Packages Structure

Name	Level	Type	Encapsulates
Coordinate_Vector_ Matrix_Algebra	TLCSC	package	following 4 generic packages and 3 generic functions
Vector_Operations	LLCSC	generic package	type *Vectors*, and 8 functions, e.g., "+", "-", Dot_Product
Matrix_Operations	LLCSC	generic package	6 functions
Vector_Scalar_ Operations	LLCSC	generic package	3 functions
Matrix_Scalar_ Operations	LLCSC	generic package	2 functions
Cross_Product	unit	generic function	
Matrix_Vector_Multiply	unit	generic function	
Matrix_Matrix_Multiply	unit	generic function	

The Coordinate_Vector_Matrix_Algebra TLCSC of Table 2-1 is a package of generic packages and generic functions which define and/or operate on coordinate vectors and matrices. It is not a part. The four LLCSCs within it are not parts either, but the 19 functions encapsulated within them are all parts. Finally, the remaining three "miscellaneous functions" (e.g., Cross_Product) are also parts. Thus, the total number of parts within the TLCSC is 22.

As a result of this structure, CAMP entities (parts and non-parts within the Parts Catalog) have both explicit and implicit contextual dependencies. A CAMP entity may explicitly depend on other CAMP entities through the Ada context clause (i.e., the Ada with statement). In fact, most of the CAMP entities are with-dependent on other entities. For example, the Coordinate_Vector_Matrix_Algebra package (from above) explicitly depends

upon two packages (General_Purpose_Math and Polynomials), neither of which are parts.

In addition, a CAMP entity may also require types or objects that have been encapsulated with it, but not within it. These are implied contextual dependencies. For example, there are 8 functions encapsulated within the Vector_Operations LLCSC; each is a part. The functions require the data type *Vectors*, also defined within the LLCSC Vector_Operations, and are therefor implicitly dependent upon the LLCSC, which is not a part.

In summary, numerous organizational packages exist within the CAMP Parts Catalog which are not parts. Therefore, through explicit and implicit dependencies, CAMP entities frequently rely on other parts, and even non-parts.

The CAMP developers concluded that their Parts Catalog approach had organizational deficiencies, and that it also resulted in too many entities. As a result, they recommended cataloging logical parts instead of physical Ada code units. We feel that this is primarily a parts retrieval issue, and that Ada code units may make good parts. Logical units probably make better parts and should be in the Parts Catalog as well. Furthermore, we believe (in contrast to the CAMP criteria) that organizational packages (e.g., Coordinate_Vector_Matrix_Algebra) should be parts, thus serving to eliminate implied contextual dependencies.

As for too many entities, a better approach may be to explicitly represent nested parts in a hierarchy. One way to do this would be to classify the parts by their level (i.e., TLCSC, LLCSC, unit). While this would not decrease the number of parts, it would provide for their effective management (i.e., improving access form logical to physical parts). For example, if the LLCSC package Vector_Operations was identified as a part, then the 8 functions within it may be identified as nested parts, each with an explicit dependency on their encapsulating part.

In summary, the CAMP parts can generally be characterized as:

- strongly typed (floating point based)
- highly inter-dependent
- built upon low-level encapsulation of data types and operations
- highly generic (some very complex)
- supporting complex Ada generics through pre-defined default generic parameters (including subprograms)

The CAMP developers used the term *semi-abstract* to describe their parts composition approach to developing reusable Ada parts for RTE applications. The **semi-abstract** approach is a part design method based on Ada generics in

which a part is designed to provide the user with both an abstract interface and a mechanism for directly accessing the internal structure of the part. The semi-abstract method yields a combination of high-level generic parts with lower level support packages providing actual types and operations, leading to the creation of a complete environment for use of a part. These pre-defined structures of high-level parts with supporting lower level parts are called **bundles**. The semi-abstract approach also provides the user with an open architecture — the ability of the user to supply his own data types and operators. Thus, the user may tailor a bundle, overriding aspects of the bundle by supplying other CAMP parts or his own parts.

Actually, bundles are informal schematics of high-level subsystems. In fact, there are a set of CAMP constructors which facilitate the composition of an entire subsystem based on a bundle of CAMP parts. Figure 2-3 [MCD87a] illustrates the composition of an autopilot subsystem, and depicts the underlying CAMP parts. These concepts will be discussed in more detail in the following section.

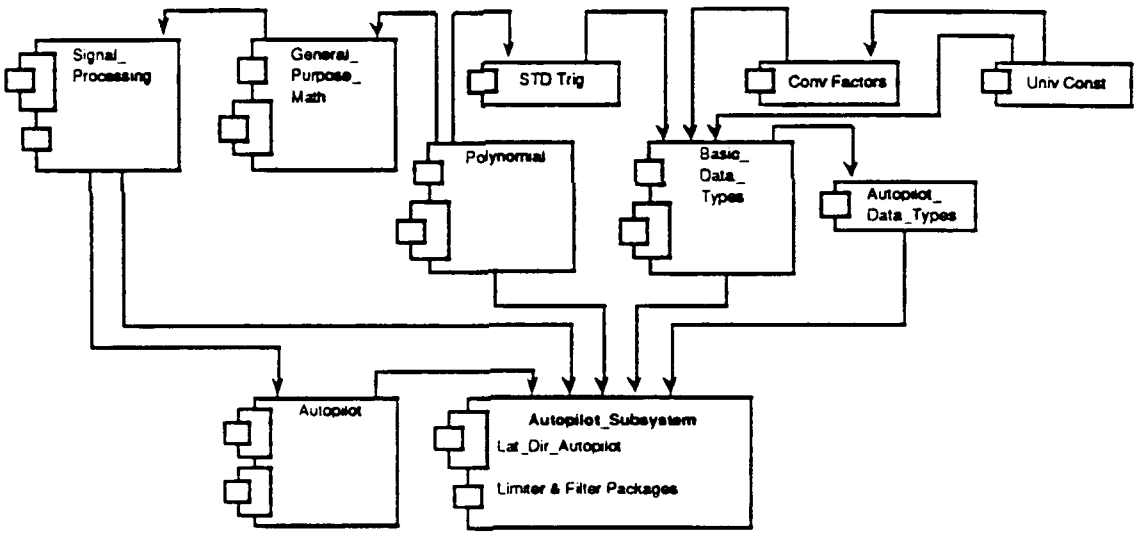


Figure 2-3 Assembling [Composing] an Autopilot Application

2.2.3 Analysis of the CAMP Component Constructors

The Component Construction subsystem of the AMPEE prototype parts composition system is composed of a set of 12 component constructors. Each of the constructors is identified and briefly described below:

- **Kalman Filter** — supports the construction of a tailored Kalman filter subsystem, including the data types and operations needed to support Kalman filter operations

- **Pitch Autopilot** — supports the construction of a tailored pitch autopilot subsystem, including the required data types, filters, and a limiter
- **Lateral/Directional Autopilot** — supports the construction of a tailored lateral/direction autopilot subsystem, including the required data types, filters, and limiters
- **Navigation Component** — supports the construction of a set of individually tailored navigation computation components
- **Navigation Subsystem** — supports the construction of a tailored navigation subsystem composed of a set of integrated navigation computation components
- **Finite State Machine** — supports the construction of a tailored component which implements a finite state machine, including the Mealy and Moore varieties
- **Data Bus Interface** — supports the construction of a tailored component which provides a general-purpose interface to a data bus
- **Data Type** — supports the construction of a tailored package of Ada data types, including discrete types, arrays, records, and access types
- **Time-Driven Sequencer** — supports the construction of a tailored sequencer which implements a time-driven sequence of actions
- **Event-Driven Sequencer** — supports the construction of a tailored sequencer which manages an ordered set of events whose occurrence is prerequisite to some action
- **Task Shell** — supports the construction of a set of individually tailored task shells which implement asynchronous process (e.g., periodic, continuous, data driven, or interrupt driven)
- **Process Controller** — supports the construction of a tailored process controller which manages an integrated set of asynchronous processes (i.e., task shells)

All of the CAMP constructors can be described as schematic constructors. The common model of CAMP schematic construction is illustrated in Figure 2–4.

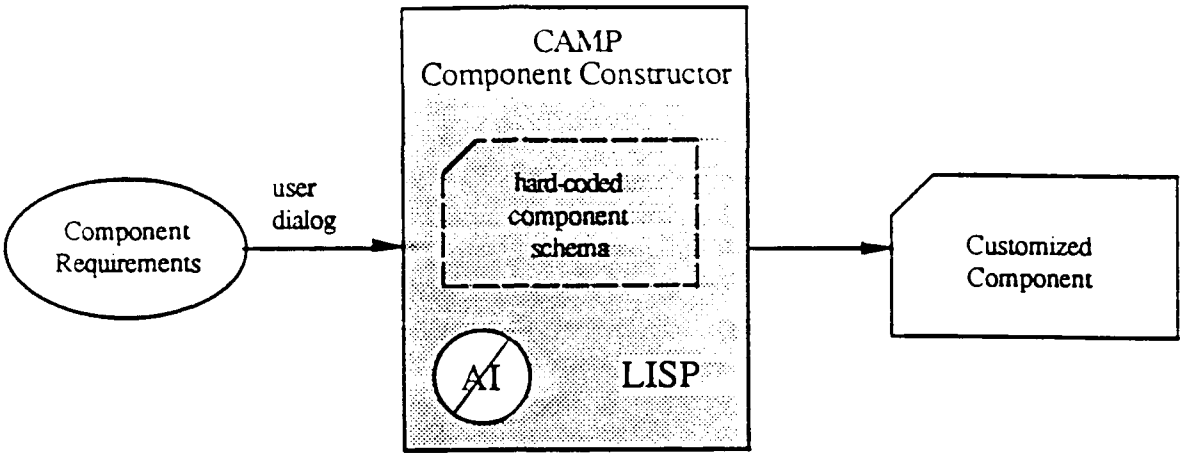


Figure 2–4 CAMP Schematic Construction

As mentioned previously, what has been referred to as CAMP *meta-parts* and *schematic parts* do not exist within the CAMP Parts Catalog, or anywhere else. Looking back at Figure 2–2, it appears as though the CAMP developers originally intended to represent schematic parts within an expert system knowledge base, along with a set of construction rules used to generate code from the schematic part. However, as shown (by omission) in Figure 2–4, our analysis has uncovered no significant application of AI technology within the CAMP constructors. On the other hand, each CAMP constructor does contain—in the essence of its design—the equivalent of a schematic part. This is depicted in Figure 2–4 as the **Component Schema.**

We wish to strongly emphasize that there is no data structure, file, knowledge base, or other realization which can be associated with the term component schema. The same is especially true for *meta-part* and *schematic part*, since these terms implied that they were *parts*. Likewise, be wary of the terms *blueprint, template, decision tree, skeleton,* and *bundle*. All of these terms refer to nothing more than the hard-coded knowledge incorporated into the design and implementation of the constructors.

The CAMP constructors can be described as context-sensitive template-driven code generators. The templates are hard-coded within the constructors. They are complex, not like simple fill-in-the-blank templates. As modeled in Figure 2–4, the schematic constructors elicit necessary **Component Requirements** from the user, typically through a dialog, and then generate the corresponding **Customized Component**. The component schemas can also be thought of as decision trees which drive the dialog in response to user inputs. The specification technique utilized for the component requirements dialog is unique for each constructor, depending on the nature of the underlying component. After all the necessary

requirements have been input, the customized Ada component is generated and provided for the user.

Besides the customized Ada component, the CAMP constructors generally produce two additional kinds of output: a *requirements set* and a *help file*. The **help file** contains the names of all the files and Ada program units generated by the constructor, and any other information the user may need to locate and use the generated component. The **requirements set** is an internal representation of the component requirements used to generate the customized component. By storing the requirements sets, the constructors are able to provide the user with the ability to modify previously entered requirements and regenerate software components. The user is also provided with the ability to delete existing requirements when they are no longer needed. "Note that in some of the constructors the *modify* options are not functional." [MCD87b]

The CAMP developers have classified the CAMP constructors as *generic instantiators* and *schematic part constructors*. Some constructors are said to combine elements of both types (e.g., Kalman Filter Constructor). In addition to this, the CAMP documentation eludes to a general-purpose Generic Instantiator Constructor in several places, including a draft version of the AMPEE detailed design document. [MCD86] However, to the best of our knowledge, such a constructor does not exist. Furthermore, we feel that the generic instantiator classification does not accurately portray the constructors in this category.

In our opinion, the CAMP schematic constructors generally fall into two classes: *subsystem composition* and *component generation*. Figure 2–5 illustrates our classification of the CAMP constructors. The subsystem composition constructors are the Kalman Filter Constructor, the autopilot constructors, and the navigation constructors. The remaining CAMP constructors are component generation constructors. The two types of constructors are alike in that they are both a form of schematic construction; they differ in their level of abstraction.

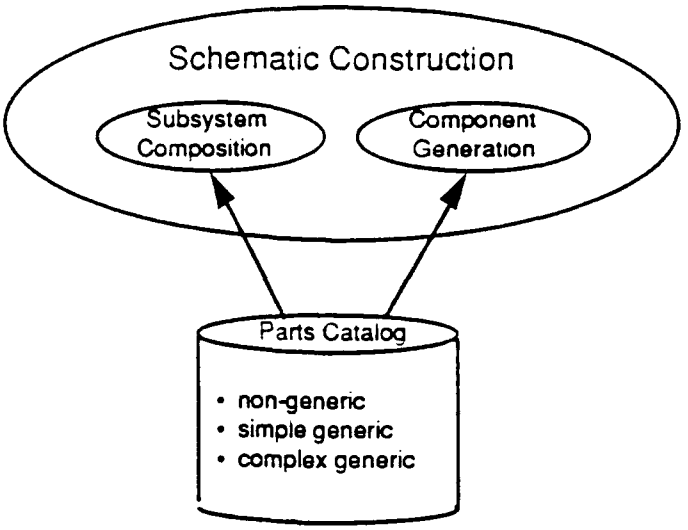


Figure 2–5 Classes of Schematic Construction

Component generation is the schematic construction of a customized Ada component. Component generation constructors generate low-level Ada code (e.g., subprogram bodies), and do not rely heavily on parts. **Subsystem composition** is the schematic construction of a customized subsystem composed of numerous Ada parts. Subsystem composition constructors generate high-level Ada code dealing with the package structure of the subsystem. To illustrate the differences between these two types of schematic construction, Figures 2–6 and 2–7 provide an example from each category.

```
with   Basic_Data_Types, Autopilot_Data_Types, Signal_Processing, Autopilot;
use    Autopilot_Data_Types;

package <PA Component Package Name> is

    package ADT renames Autopilot_Data_Types;

    – Data Subtypes
    subtype PA_Fin_Deflections is ADT.<Fin Deflection Customization>;
    ...

    – Elevator Command Limiter
    package Command_Limiter is new Signal_Processing.<Limiter Instantiation>;

    -- Acceleration Filter
    package Acceleration_Filter is new Signal_Processing.<Digital Filter Instantiation>;

    -- Pitch Rate Filter
    package Pitch_Rate_Filter is new Signal_Processing.<Digital Filter Instantiation>;

    – Pitch Autopilot
    package Pitch_Command is new Autopilot.<Pitch Autopilot Instantiation>;

end <PA Component Package Name>;
```

Figure 2–6 Example Subsystem Composition

As shown in Figure 2–6, the Pitch Autopilot Constructor generates an Ada package specification, and its associated context and use clauses. By utilizing the underlying CAMP parts, the constructor is able to compose a complete pitch autopilot subsystem of a missile application. The example assumes that the user has selected to use the Autopilot_Data_Types part, as well as parts within the Signal_Processing package (to instantiate a limiter and 2 digital filters). The purpose of the Autopilot_Data_Types subtypes is to allow range and accuracy customization of the base autopilot data types. The limiter and filter instantiations depend upon the customized data types and their associated operations, as well as the customized limiter and filters. Finally, the instantiation of the Pitch_Autopilot part — a complex generic with more than a dozen generic parameters — completes the composition of the pitch autopilot subsystem.

```
with   <Action Packages>;

package <FSM Component Package Name> is

    type States is (<States>);

    type Stimuli is (<Stimuli>);

    function Current_State return States;

    procedure Signal (Event: in Stimuli);

    Invalid_Stimuli : EXCEPTION;

end <FSM Component Package Name>;

-------------------------------------------------------------------------------

package body <FSM Component Package Name> is

    Present_State : States := <Initial_State>;

    Event : Stimuli;

    function Current_State return States is
    begin
        return Present_State;
    end Current_State;

    procedure Signal (Event: in Stimuli) is
    begin
        <State Transitions>
    end Signal;

end <FSM Component Package Name>;
```

Figure 2–7 Example Component Generation

The package specification and body of Figure 2–7 illustrate the particular Ada implementation of a finite state machine (FSM) which the Finite State Machine Constructor provides. No other CAMP parts are referenced by the generated component. The user provides the FSM's states, and transitions between states. Transitions are represented as an ordered pair of states, and one or more causal stimuli. The user also specifies an initial state, and any possible actions which may be associated with transitions (Mealy machine), or states (Moore machine). Actions are represented as parameterless procedures (this is common throughout the constructors). If any actions are specified, their encapsulating packages must be provided for the context clause. The

user is provided with a function which returns the FSM's current state, and a procedure which inputs a stimulus and processes it with respect to the current state. The FSM constructor also checks the component requirements for anomalies and optimizes the customized component for time and space efficiency.

As suggested by the pitch autopilot example, generic instantiation plays a major role within subsystem composition constructors. In this sense, it can be construed that we are merely suggesting another name for the CAMP *generic instantiation* constructors.

While subsystem composition does involve a great deal of generic instantiation, it is *primarily* a form of schematic construction, as is component generation. Constructors of both types access generic parts from the Parts Catalog, and perform generic instantiation. Therefore, generic instantiation should not be used to distinguish between the two types of construction. Furthermore, we feel that *subsystem composition* and *component generation* are more indicative of their function, especially in terms of reuse.

Table 2–2 identifies the major characteristics of each of the CAMP constructors, specifying their *type* of schematic construction, as well as their CAMP and domain dependencies. A constructor is **CAMP-dependent** if it depends on CAMP parts, packages, or other CAMP constructors. A constructor is **domain-specific** if it has negligible utility outside of the missile flight software domain.

Table 2–2 Characteristics of the CAMP Component Constructors

Component Name	Subsystem Composition	Component Generation	CAMP-Dependent	Domain-Specific
Kalman Filter	•	•	•	•
Pitch Autopilot	•		•	•
Lateral/Directional Autopilot	•		•	•
Navigation Component	•		•	•
Navigation Subsystem	•		•	•
Finite State Machine		•		
Data Bus Interface		•	•	
Data Type		•	•	
Time-Driven Sequencer		•	•	
Event-Driven Sequencer		•		
Task Shell		•	•	
Process Controller		•	•	

The **Kalman Filter Constructor** is primarily a subsystem composition constructor. However, embedded within it is the component generation construction of (statically and dynamically) sparse matrix components. The Kalman Filter Constructor is dependent on numerous CAMP parts, and generates a Kalman filter subsystem which is specific to missile systems.

The **Pitch Autopilot Constructor** and the **Lateral/Directional Autopilot Constructor** are separate subsystem composition constructors that share much of the same user interface and processing. [MCD87b] These constructors both depend on numerous CAMP parts and generate autopilot subsystems which are specific to missile systems.

The **Navigation Component Constructor** and the **Navigation Subsystem Constructor** are also separate subsystem composition constructors that share much of the same user interface and processing. [MCD87b] These constructors both depend on numerous CAMP parts and generate navigation computation components specific to missile systems (e.g., missile velocity and position computations).

The **Finite State Machine Constructor** is a component generation constructor. As shown earlier, it is not CAMP-dependent; nor is it domain-specific. In fact, this constructor is potentially applicable to any domain.

The **Data Bus Interface Constructor** is a component generation constructor which also performs some low-level generic instantiation. The constructor generates an integrated set of package specifications and bodies consisting of numerous data types, objects, and subprograms which facilitate the sending and receiving of messages. The constructor is dependent on the CAMP generic queue parts, which are customized by the user (e.g., bounded or unbounded FIFO buffer, circular buffer). The package bodies generated by the constructor contain application-specific subprograms with null bodies (i.e., code templates). The subprogram bodies must be completed by the user. This constructor is not domain-specific, and may be applied to numerous domains which favor distributed systems.

The **Data Type Constructor** is a low-level component generation constructor. It may be used to generate a package specification of various Ada types. The constructor is dependent on the Basic_Data_Types (BDT) part, and other CAMP packages. It utilizes these packages to facilitate the definition of customized floating point types and special-purpose array (e.g., vector, matrix) types and operations. The constructor is applicable to any domain.

The Data Type Constructor is also utilized by other constructors (e.g., Kalman filter, autopilot, navigation, data bus interface) to allow definition of needed types: "Other constructors incorporate parts of the Data Types Constructor..." It should also be noted that the constructor "currently provides capabilities only to define scalar types (i.e., discrete and floating point types)." [MCD87b]

The **Time-Driven Sequencer Constructor** and the **Event-Driven Sequencer Constructor** are separate component generation constructors which generate general-purpose sequencers. The Time-Driven Sequencer Constructor generates a package specification and body that performs the time-driven sequencing specified by the user. In order to use the part, the user must declare an object of the type *Delays*, defined by the constructor; the data object supplies the delays which are used to drive the sequencer. This constructor is dependent upon the Clock_Handler CAMP part. The Event-Driven Sequencer Constructor generates a package specification and body with an

embedded task that manages the event-driven sequencing specified by the user. This constructor is not CAMP-dependent. Neither constructor is domain-specific.

The **Task Shell Constructor** and **Process Controller Constructor** are separate component generation constructors which generate Ada tasks for specified asynchronous processes. Both constructors are dependent upon and instantiate task shells (i.e., generic packages with embedded tasks). It is assumed that these constructors share much of the same user interface and processing. The Task Shell Constructor generates a set of independent task shell instantiations. The Process Controller Constructor generates a package specification and body with embedded task shell instantiations, as well as an inter-task coordination procedure which controls task activation, execution, and termination. Neither constructor is domain-specific.

In summary, subsystem composition constructors, as used in CAMP, generally have the following characteristics:

- provide automated assistance with the customized composition of a pre-defined structure of CAMP parts
- perform simple and complex generic instantiation
- exhibit medium to high complexity
- depend on numerous CAMP parts
- apply only to the missile flight software domain

In contrast, component generation constructors, as used in CAMP, generally have the following characteristics:

- facilitate the automated generation of customized Ada program units (including code bodies)
- perform simple generic instantiation
- exhibit low to medium complexity
- depend on few CAMP parts
- can be applied to many domains

In conclusion, we believe that both types of schematic construction promote reuse and increase productivity. However, we feel that the CAMP subsystem composition constructors are less attractive than the CAMP component generation constructors. In comparison to the component generation constructors, the subsystem composition constructors are generally less cost-effective because they are:

- more expensive to develop due to relative complexity
- more difficult to maintain due to part coupling

- more domain-specific

The component generation constructors do not suffer as much from the drawbacks of the subsystem composition constructors. There are, however, some CAMP-dependencies within these constructors which may lead to maintenance problems. Also, since these constructors are mostly domain-independent, their modifiability/extensibility is an important concern. These issues will be addressed in the following section.

2.2.4 Modification Required to Support Another Domain

According to CAMP, the characteristics of a domain will impact what parts are designed, how the parts are designed, and also how the parts fit together. Characteristics of the missile systems domain which affected the CAMP parts and constructors include [MCN86a]:

- Very high degree of data flow inter-connectivity
- Complex decision-making
- Large number of mathematical data transformations
- Little data movement
- Relatively simple data structures
- External interfaces with special purpose equipment
- Processes that have rigid temporal relationships
- High use of intermediate results of calculations
- Asynchronous time-driven processes

Compare these characteristics to some of those representative of C^2 systems [QUA88]:

- Support for distributed operations
- Substantial amount of parallel processing
- Data-driven approach to function implementation
- Management and transfer of large amounts of data
- Data analysis and decision support
- Management of secure data
- Relatively less stringent performance requirements
- Asynchronous event-driven processes
- Trend to COTS hardware and software

The amounts and kinds of data transfers between parts in this domain would be very different from CAMP parts. The process scheduling mechanisms for this domain are also generally very different. Distribution and parallelism would influence the partitioning of parts and their interaction mechanisms, although the newer missile systems also must address these issues. Efficiency of the generated code (both in terms of timing and sizing of parts) is not a major consideration.

After having analyzed the CAMP parts and Parts Composition System (PCS), and having considered the implications of the C^2 domain, we are now ready to draw conclusions and provide recommendations regarding potential PCS support within other domains, especially C^2. Specifically, we will address the existing CAMP constructors, as well as constructor candidates which may serve to extend the domain-independence of the PCS.

The five CAMP subsystem composition constructors seem to have the least potential benefit in terms of modifiability and support of another domain. To begin with, these constructors are primarily domain-specific, as a result of their underlying subsystems (parts). For example, a CAMP Kalman filter subsystem may have some potential in an avionics application, but is probably not applicable to a domain such as C^2.

For the sake of analysis, lets assume that one of the subsystem composition constructors is applicable to some other domain of interest. Of course, this implies that the constructor's underlying parts are applicable to the new domain. If the parts are directly applicable (i.e., requiring no modification) or only slightly altered, then the constructor would most likely require little or no modification as well. If the parts are substantially altered to accommodate the new domain, then a decision is to be made on whether or not to modify the constructor too. Under such circumstances, if the subsystem is not expected to be reused a great deal, then modifying the constructor would not be cost effective, since the subsystem can be composed manually.

The seven CAMP component generation constructors, being primarily domain-independent, have a much higher potential for reuse in other domains. Some, however, do have CAMP dependencies. In this case, modification may be desirable in order to eliminate future maintenance problems. Other modifications may be desirable as well. For example, to enhance their capabilities, or to alter their underlying component schema (e.g., algorithm modification).

Finally, it may be possible to extract domain-independent constructors from within complex CAMP domain-specific constructors. There are three such constructor candidates that have been identified:

1) Matrix Constructor
2) Digital Filter Constructor
3) Integrator/Limiter Constructor

Actually, the Matrix Constructor was identified as a separate constructor in earlier CAMP documentation. [MCD85a] It may be cost effective to extract it from within the Kalman Filter Constructor and perform the modifications necessary to make it an independent constructor.

Likewise, a Digital Filter Constructor and/or an Integrator/Limiter Constructor may be extracted from the CAMP autopilot constructors. It appears as though the processing within the autopilot constructors would facilitate this extraction.

2.2.5 Adaptations within CAMP

One measure of the goodness of a reusable software part is its ability to adapt to a wide range of user requirements. By being able to adapt in this fashion, a part stands a greater chance of being reused.

An **adaptation requirement** is a statement specifying a particular kind of adaptation expected of a part. Adaptation requirements are independent of the design and implementation of a part, and are stated in terms of the part's requirements. Low-level adaptation requirements are generally concerned with programming primitives (e.g., data, operations). Higher level adaptation requirements are generally related to the part's environment (e.g., number of users, external interfaces).

An **adaptation mechanism** is a means by which a part facilitates an adaptation requirement. For a given adaptation requirement, there may exist several possible adaptation mechanisms.

For example, consider an Ada part which provides a stack object and operations. An obvious adaptation requirement for a stack part is the ability to adapt to different data types for the stack elements, since stack operations are not dependent on the element type. Naturally, the Ada generic facility is the logical adaptation mechanism for this requirement. In particular, the formal generic type parameter is the mechanism which could facilitate this adaptation requirement. Thus, a generic Ada part could be developed which utilizes a formal generic type parameter as an adaptation mechanism to satisfy this adaptation requirement.

There are, however, some adaptation requirements for which the Ada language provides no suitable adaptation mechanism. For these adaptation requirements, automated assistance is necessary to provide adaptable Ada

parts. Thus, by identifying the adaptation requirements and mechanisms which do and don't require automated assistance, we will gain a better understanding of the requirements of the Generalized Constructor Capability, and more importantly, reuse engineering in general.

Analyzing the CAMP parts and constructors in this perspective has taught us that separating adaptation requirements from adaptation mechanisms is very complex, and difficult. However, we believe that doing this is essential to the specification and engineering of reusable software parts, and is a critical aspect of this research effort.

From our investigation of the CAMP parts and constructors, we have identified the following low-level adaptation requirements:

- algorithm selection (from pre-determined alternatives)
- data type selection (from pre-determined alternatives; implying an underlying algorithm selection)
- personalization of identifiers within components
- customization of data types (range, precision)
- *default* object initialization values

And the following adaptation mechanisms:

- *default* generic subprogram parameters (Many CAMP generic parts have formal generic subprogram parameters. For each such part, there are lower level support packages which provide *default* subprograms for these generic parameters. By specifying the support package in the appropriate with clause, a convention has been followed which provides this adaptation mechanism.)
- the specification of parameterless procedures for *actions* — similar to procedural data types

As our research progresses, we will continue to identify adaptation requirements and mechanisms, and attempt to classify them and understand their implications on reuse and the GCC.

2.3 CAMP – Phase 3

The CAMP – Phase 3 (CAMP–3) project is approximately a two-and-a-half year effort scheduled for completion in early 1991. Three of the known objectives of the CAMP–3 effort are to update the CAMP parts; re-engineer the Parts Composition System (PCS) in Ada; and continue the component construction research and development.

In January of 1990, we received a copy of the Parts Engineering System (PES) catalog developed on the CAMP–3 project. This system is the result of the PCS re-engineering effort mentioned above. Included in the shipment was a PES User's Guide. As described by McDonnell Douglas:

> One of the problems associated with software reuse in the past has been the lack of information about reusable parts. The PES catalog is designed to alleviate this problem by providing the user with a means to obtain information on the availability and relevance of reusable software parts. The PES catalog is intended for use by both software engineers and application engineers.
>
> The PES catalog is based on the Ada Missile Parts Engineering Expert (AMPEE) system catalog developed during CAMP–2. The AMPEE system catalog was written in LISP and ART, and was hosted on a Symbolics 3620. The PES catalog is the result of a re-engineering effort to create a catalog with enhanced capabilities, written in Ada, and hosted on a VAX running VMS. [PES shipment letter, December 1989]

From reviewing the PES User's Guide, it appears that the catalog system is very much improved. The following list identifies the major *enhanced capabilities*:

- interactive, batch, and callable interfaces
- multiple domain and project capacities
- improved query and search capabilities
- additional and improved management of catalog entities such as project names, domain taxonomies, keywords, etc.
- provisions for site-specific tailorability and extensibility

Of particular interest is the new classification of part types. The PES catalog User's Guide identifies the following four part types [MCD89]:

1. Operational — the reusable functions and procedures, and "occasionally" packages, that can be incorporated into a user's application code
2. Bundle — an entity that contains operational parts or other bundles
3. Constructor — a software system that facilitates the development of application software by producing software components based on user requirements; each constructor is based on either a complex Ada generic unit or a schematic part
4. Schematic — a "*part blueprint* or template of the part's structure"

This new and improved parts classification scheme gives much credence to our analysis of the CAMP parts and constructors (see Sections 2.2.2 and

2.2.3). Notably, there is an absence of any "generic" types, and there is the introduction of the "bundle" type. We are still skeptical of the "schematic" type. It is interesting to note, however, that constructors themselves are now considered to be parts. Other than this, it is yet unknown what the CAMP–3 developers have accomplished in terms of component construction.

3 Generalized Construction Approaches

One of the key results of the CAMP project (phases 1 and 2) was the development of a prototype parts composition system, including a set of Ada component constructors. It has been shown that some of these constructors (almost half) generate components which serve to compose large, complex subsystems from a predefined, adaptable bundle of CAMP parts. Other CAMP constructors generate lower level components which are primarily stand-alone units. Our research outside of the CAMP arena has provided us with a better understanding of these two types of constructors. As a result, we have reached the following conclusions:

1) There will always be some form of reusable software "parts," driving the need for comparable **software adaptation and composition** facilities.

2) The repeated application of a programming language to similar, "difficult" problems will always drive the need for advanced **software construction** facilities.

This section of the report—Generalized Construction Approaches—will cover the research we have done outside of CAMP (see Figure 3–1), providing much supporting evidence of our conclusions. Our research has provided us with the insight that *constructors are a part of the natural evolution of programming languages,* and that we must take many factors into consideration before allocating resources to their development.

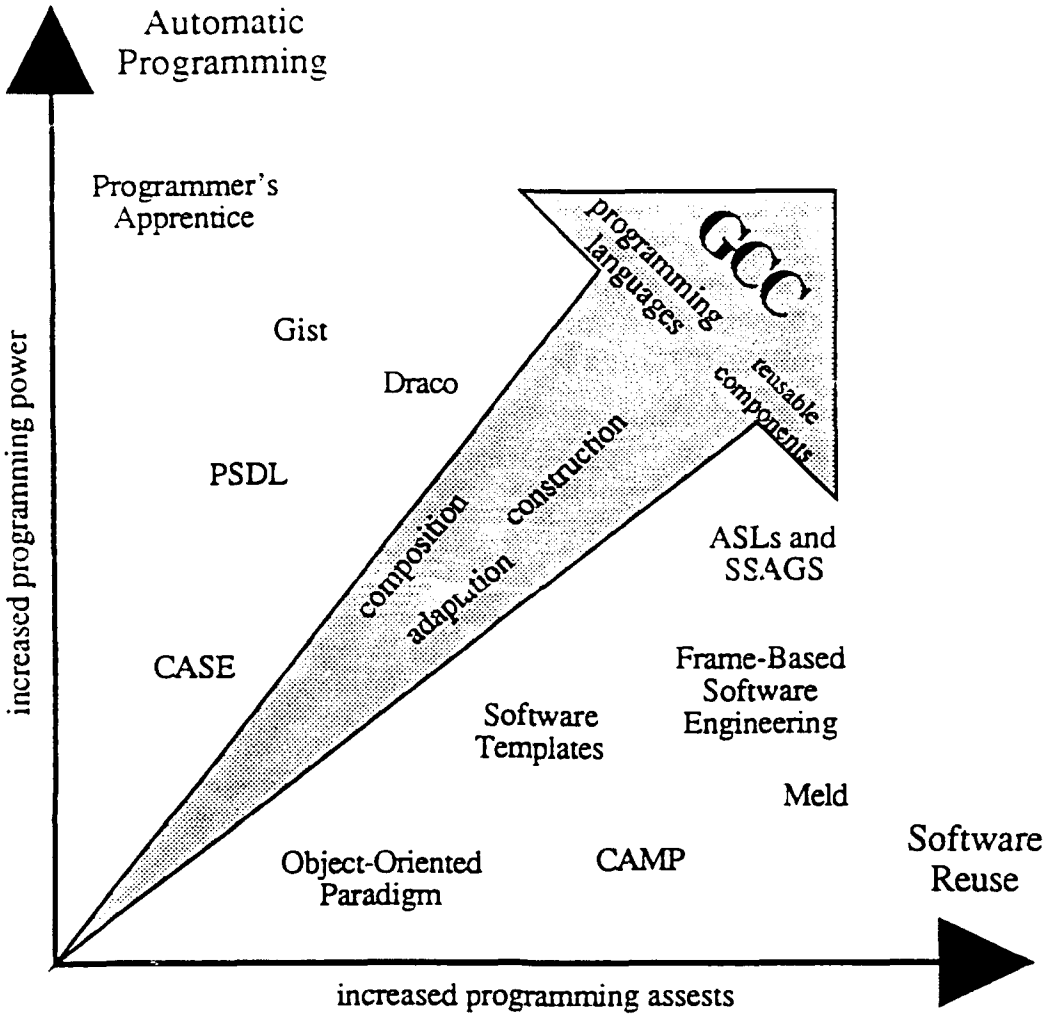


Figure 3–1 Research Summary—Gaining a New Perspective

Working definitions of several relevant and driving terms are now in order:

A software **part** or **component** is a reusable software entity. Components are typically cataloged in a reuse library and made available for browsing and/or retrieval. Any reusable software entity may be considered a software component (i.e., not just program code).

Software **composition** is the piecing together of software components into a larger software system. (Note: composition is a form of construction, to be defined below.) Software **composition mechanisms** are the various means—manual or automated—in which components may be connected. Software **composition models** represent the characteristics and structure of the glue which holds together and connects the components of a software system.

A software component often must be customized or modified before it can be reused within some application. This form of preparation is called **adaptation**. A software component which has been engineered for reuse should be able to adapt easily to different requirements; within reason. This ease of adaptation is called **adaptability**. Engineered adaptability is specified by **adaptation requirements** and are facilitated by **adaptation mechanisms**. Tailorability, extensibility and portability are all forms of adaptability. Abstraction also promotes adaptability. As advocated by Biggerstaff and Richter of the Microelectronics and Computer Technology Corporation, we too believe that adaptation and its automation are key to software reuse:

> The modifying process is the lifeblood of reusability. It changes the perception of a reusability system from a static library of rock-like building blocks to a living system of components that spawn, change, and evolve new components with the changing requirements of their environment.
>
> It is overly optimistic to expect that we can build a reusability system that allows significant reuse without the need to modify some portion of the components. However, modification is largely a human domain. There are few tools that provide any measure of help in modifying components. [BIG87]

Software **construction** and **generation** are terms used to describe the *production* of software products—primarily program code. In a general sense, they are synonymous. However, construction may be manual or automated, whereas generation implies automation. In addition, generation usually implies the production of a relatively large body of software (e.g., application *generation* vs. component *construction*). One aspect of software construction—of primary significance—is reuse; the utilization of previously developed software components within the construction of a larger software system.

Software **construction mechanisms** are the facilities provided by a programming environment which specifically support the production of software. As with composition mechanisms, construction mechanisms may be manual or automated. Examples of automated construction mechanisms are the macro facilities of assembly languages, and the Ada generic instantiation facility. It is evident, especially from the Ada generic example, that construction mechanisms and adaptation mechanisms are closely related and have overlapping domains.

A software **construction technique** is the systematic application of a set of software composition and construction mechanisms. These mechanisms may be manual and/or automated. A construction technique is therefore the active sense of a composition model. For any given software construction technique, the scope of construction may be a single component (e.g., algorithm, subprogram, ADT, object), a subsystem of an application, an entire

application, all applications within a specific domain, or within the entire omniverse.

Obviously, software composition and construction are very much related. In fact, they must be synergistic to be effective: *a component must be consistent with a software system's model of composition in order to be efficiently composed within the system.* In other words, reusable software components may be useless—as is—in a software system utilizing significantly different composition mechanisms. In this respect, composition models are of primary importance when considering automated construction approaches. In light of this, we have taken a step back from the CAMP work to assess the current technology of software composition and construction.

3.1 Software Composition and Construction Technology Assessment

This section provides an assessment of the current technology of software composition and construction. Included in this section are many references to previous technology, providing an evolutionary perspective.

The software composition assessment covers the spectrum of software composition mechanisms, proceeding from lower to higher level mechanisms.

The software construction assessment includes many research projects (identified in Figure 3–1), as well as low-level construction mechanisms. This section proceeds from the larger research efforts down through the construction mechanisms.

3.1.1 Software Composition Mechanisms

Assembly-level composition mechanisms include interrupts, subroutines (call and return), and various forms of branching (e.g., conditional, unconditional, direct, indirect). Most high-level programming environments provide analogous low-level composition mechanisms and a set of higher level composition mechanisms which—as a whole—imply a nominal composition model for that environment. The remainder of this section presents several higher level composition mechanisms.

The UNIX pipe mechanism [KER84] is a frequently cited example of **serial or dataflow** composition. Serial composition is a very weak form of composition—the components are connected only in the sense that they are executed one after the other, with the output of one component being the input of the next.

Procedural composition is typified by PL/I and Pascal. Procedural invocation is based on the assembly-level subroutine concept, with variations concerning formal and actual parameters and recursion.

Functional composition builds upon procedural composition with the addition of a returned value. This allows function invocations to be nested within expressions. LISP and 'C' strongly advocate the functional model of composition.

Modular composition supports the encapsulation of data types, objects, and operations into a single program unit. Modula–2 modules and Ada packages are good examples of such composition mechanisms. Modules may also support information hiding via separate interface specifications and implementations. Both Modula–2 modules and Ada packages provide this composition mechanism as well.

Modules may be utilized to create Abstract Data Types (ADTs) which provide a data type and the operations available for objects of that type. The object-oriented paradigm is based on the concept of ADTs. Within the object-oriented paradigm, a single program structure is both an ADT and a *type*. This structure was originally called a *class* in Simula 67—the pioneer object-oriented programming language. [MEY87]

The object-oriented paradigm provides several composition mechanisms for objects (i.e., instances of a class type). The most obvious is **message passing**, somewhat similar to the procedural composition mechanism. However, dynamic binding allows message passing to be much more flexible. **Inheritance** is another powerful composition mechanism, tying together classes (and indirectly, objects) into an inheritance hierarchy.

Constraints provide a powerful method of connecting components. Simply stated, a constraint is an "enforced relationship." [EGE89] As implied by Leler, the object-oriented paradigm provides an appropriate foundation for constraints:

> In constraint languages, programming is a *declarative* task. The programmer states a set of *relations* between a set of *objects*, and it is the job of the constraint-satisfaction system to find a solution that satisfies these relations. [LEL88]

Constraint satisfaction involves problem solving. As a result, constraints have long been a part of AI research (e.g., Sketchpad, Steele's Constraint Language, TK!Solver). In addition, constraints have been applied to various software systems—mostly object-oriented—for their expressiveness and power, and their ability to represent behavioral specifications (e.g., ThingLab).

Two additional examples—discussed in the following section—are Gist and Meld.

Another mechanism rooted in AI which can be used to compose components is **logical inferencing**. Logic programming languages (e.g., Prolog), theorem proving systems, and some applications of knowledge representation (e.g., expert systems), all utilize some form of logical inferencing based on facts or assertions and logic rules. Expert systems—which combine knowledge bases, facts, rules and inference engines—have become very popular, even commercially, due to their automated reasoning capabilities. In a programming environment, facts and rules can be used to invoke components, providing a powerful component composition mechanism.

3.1.2 Software Construction Research

In a sense, *automatic programming* represents the ultimate form of software construction. Over the years, the term **automatic programming** has come to represent the elusive goal of a kinder, gentler world in which:

> There will be no more programming. The end user, who only needs to know about the application domain, will write a brief requirement for what is wanted. The automatic programming system, which only needs to know about programming, will produce an efficient program satisfying the requirement. Automatic programming systems will have three key features: They will be end-user oriented, communicating directly with end users; they will be general purpose, working as well in one domain as in another; and they will be fully automatic, requiring no human assistance. [RIC88a]

Judging from the current state-of-the-art software construction systems, we are a long way from realizing this dream. By categorizing the approaches being taken to this end, we can develop a better understanding of the problem at hand and the progress being made:

1) Programming Language
2) Narrow Domain
3) Transformational Implementation
4) Semi-Automated

The **programming language approach** starts with the current level of programming technology and advances from there. This approach sacrifices end-user orientation. In the past, this approach has led from machine-level to assembly-level to high-level programming. The current goal is very high level (fourth generation) general-purpose computer programming languages. The **narrow domain approach** sacrifices generality for success within limited

domains. This approach has led to the advent of application generators and is being extended to cover wider domains. The **transformational implementation approach** attempts to bridge the gap between end-user requirements and program implementation with a series of transformations. Each transformation (possibly requiring human guidance) is designed to be function and correctness preserving. The **semi-automated approach** sacrifices full automation while attempting to provide as much automated assistance as possible over as much of the software development process as possible. Much of the commercial effort in terms of integrated software development tools and CASE products fall in this category.

3.1.2.1 Programmer's Apprentice

One of the largest and most well-known automatic programming efforts is the Programmer's Apprentice project at MIT. [RIC88b] [WAT86] [WAT85] The Programmer's Apprentice is "basic research at the intersection of artificial intelligence and software engineering[,] ... a collaboration involving students, faculty, and staff over several years." [RIC88b] This automatic programming project is one of the few with an approach that can be placed in a single category—the semi-automated approach. Unlike the more conventional semi-automated efforts such as CASE, this project is driven by the *assistant* concept, viewing the automation not as a software tool, but as a capable, intelligent, cooperating agent:

> A provocative example of the assistant approach was proposed by IBM's Harlan Mills in the early 1970s. He suggested creating "chief programmer teams" by surrounding expert programmers with support staffs of human assistants, including junior programmers, documentation writers, program librarians, and so on. Productivity was thereby increased because the chief programmer could apply full effort to the most difficult parts of a given software task without getting bogged down in routine details that currently use up most of every programmer's time[!]. Experience has shown that this division of labor can be very successful. Our goal is to provide *every* programmer with a support team in the form of an intelligent computer program called the Programmer's Apprentice. [RIC88b]

Their primary goal is to have the Programmer's Apprentice eventually span software development activities from requirements acquisition and analysis through implementation. Their approach has been to develop prototypes of parts of the Apprentice, and then connect the prototypes and rebuild on the basis of what is learned. KBEmacs (Knowledge-Based editor in Emacs) is a completed prototype which covers the implementation part of the Apprentice. Work has begun on both the Requirements Apprentice and the Design Apprentice. The project also includes other investigations, outside of the prototypes, which are based on the same underlying technology. As described by Rich and Waters, their formal representation of programs and

clichés—commonly used combinations of primitive elements with familiar names—form a cornerstone of their research:

> In general, a cliché consists of roles and constraints. The roles of a cliché are the parts that vary from one occurrence of the cliché to the next. The constraints are used to specify fixed elements of structure (parts present in every occurrence), to verify that the parts that fill the roles in a particular occurrence are consistent, and to compute how to fill empty roles in a partially specified occurrence of a cliché ...
>
> Given a library of clichés, it is possible to perform many programming tasks by inspection rather than by reasoning from first principles ...
>
> Clichés and inspection methods are theoretical (and perhaps psychological) concepts. To apply these ideas, clichés need to be represented in a concrete, machine-usable form. A cornerstone of the Programmer's Apprentice is a formal representation for programs and programming clichés called the *Plan Calculus* ...
>
> To a first approximation, the Plan Calculus can be thought of as combining the representation properties of flowcharts, dataflow schemas, and abstract data types. A plan is essentially a hierarchical graph structure ... [with] a formal semantics used for reasoning. [RIC88b]

In addition, they have developed a system to support the desired reasoning capabilities of the Apprentice—reasoning about structured software objects and their properties. Cake is "a hybrid knowledge representation and reasoning system ... that we have developed and are using for all current work in the project." [RIC88b] Cake is a layered system, with each layer building on the facilities provided by the layers below it. From top to bottom, Cake is composed of the Plan Calculus, frames, algebraic reasoning, and propositional logic.

The Programmer's Apprentice project, while still in its infancy, has already resulted in noteworthy achievements. Their work in the development of clichés, the Plan Calculus, and Cake, provides many interesting and valuable lessons with respect to adaptable, reusable components, and the automation of their adaptation, composition and construction. They have successfully developed clichés and demonstrated KBEmacs' capability with both LISP and Ada. One drawback of their current approach is that the capabilities of the Apprentice is totally dependent upon its underlying requirements, design and implementation clichés. In this sense, it can be anticipated that future Apprentices will be limited by the number (and quality) of available clichés, and therefore restricted to narrow domains—as are the current prototypes. Another current drawback of the Apprentice is that the user must reference the clichés by name. These deficiencies are mutually antagonistic. In order for an Apprentice to be generally effective, it must be supported by a substantial cliché library. But, as this library develops to maturity, it will become increasingly harder for the user to effectively reference clichés by name. This is a form of the classical

reuse library retrieval problem. From an outsider's perspective, it is somewhat surprising that such a hotbed of AI research has not identified these issues for further investigation.

3.1.2.2 Gist

Another well-known automatic programming research effort is represented by the Gist formal specification language, developed by the Information Sciences Institute's (ISI) Software Sciences Division at the University of Southern California. [BAL87] [BAL85] [BAL82] The Gist effort took an approach which combined a transformational front-end with a programming language back-end:

> Automatic programming has traditionally been viewed as a compilation problem in which a formal specification is [compiled] into an implementation. At any point in time, the term has usually been reserved for optimizations which are beyond the then current state of the compiler art. Today, automatic register allocation would certainly not be classified as automatic programming, but automatic data structure or algorithm selection would be.
>
> Thus, automatic programming systems can be characterized by the types of optimizations they can handle and the range of situations in which those optimizations can be employed ...
>
> Hence, there are really two components of automatic programming: a fully automatic compiler and an interactive front-end which bridges the gap between a high-level specification and the capabilities of the automatic compiler. [BAL85]

The ISI researchers determined that the front-end must first convert from an informal high-level specification to a formal high-level specification, and then translate from high-level formal to low-level formal. But, "because a suitable high-level formal specification language did not exist ... [they] decided to embark on a 'short' detour (from which we have not yet emerged) to develop an appropriate high-level specification language. The result of this effort was the Gist language." [BAL85]

The Gist language is an object-oriented specification language, incorporating knowledge based reasoning and constraints. It is also operational, although "the evaluation of Gist specifications [is] intolerably slow." [BAL82] This inefficiency is due to the *specification freedoms* provided by the language; non-deterministic behavioral specifications are restricted to allowable behaviors by constraints. Surprisingly, the language was determined to be unreadable, which promoted additional research concerning Gist specification validation: paraphrasing, symbolic evaluation, and behavior explanation. On the transformation side, most of the research effort focused on the required technology, rather than actual transformations—and met with limited success.

Consequently, the ISI researchers began to focus on using Gist as an operational prototype—shifting from the transformational and programming language approaches to the semi-automated approach—in an effort to specify (i.e., bootstrap) an automated software development environment. By repeatedly developing improved prototypes, they have reached "the next-generation operating system." [BAL87] Apparently, the key features of the current prototypes are a persistent object-base, associative retrieval (i.e., the ability to access objects via a description of their relationships with other objects), and constraints. The constraints in these systems may extend beyond object relationships to include general object-base processing. By continuing to enhance the capabilities and efficiency of their operational specifications, they have seemingly created another approach towards automatic programming—**perpetual prototyping**. Interestingly, their current specification language is simply an extension to LISP: "a set of macros and subroutines, plus some simple extensions to the LISP evaluator." [BAL87] Thusfar, this research has evidently resulted in great success, but at a relatively low level of application software development (e.g., object-oriented user interface, electronic mail, object browser/editor). Application in increasingly larger domains and continued prototyping will provide answers to questions regarding its potential general-purpose effectiveness.

3.1.2.3 Draco

Draco is a software construction approach—accompanied by an experimental prototype—which has been developed primarily by James Neighbors while at the University of California, Irvine. [FRE87] [NEI84] In Neighbors' words: "It has been a common practice to name new computer languages after stars. Since the approach ... manipulates special-purpose languages it seemed only fitting to name it after a structure of stars, a galaxy." [NEI84] The Draco approach is mostly transformational implementation. While it is geared at software reuse rather than automatic programming, it has the flavor of automatic programming. The Draco approach is also very domain-oriented, increasing from narrow to broad domains by building upon (i.e., reusing) its smaller domain capabilities:

> The objects and operations in a domain language represent analysis information about a problem domain. Analysis information states *what* is important to model in the problem. This analysis information is *reused* every time a new program to be constructed is cast in a domain language. Further, we propose that the objects and operations from one domain language be implemented by being modeled by the objects and operations of other domain languages. These modeling connections between different domain languages represent different design possibilities for the objects and operations in the domains. Design information state *how* part of the problem is to [be] modeled. *This design information is reused each time a new program to be constructed uses one of the given design possibilities.* Eventually, the developing program must be

> modeled in a conventional executable general-purpose language. These comprise the bottom level of the domain language modelling hierarchy. [NEI84]

The Draco approach is object-oriented, but not in the sense of object-oriented programming languages—each of the components developed in Draco is essentially an abstract data type. The approach centers upon domain analysis, capturing the analysis information in reusable components, and the iterative transformation of higher level domain language specifications into lower level specifications, "until the entire specification is expressed in the desired target domain (usually one whose language is executable) or appropriate refinements cannot be found ..." [FRE87] Peter Freeman, also from the University of California, Irvine, provides the following description of Draco and its capabilities:

> Draco is intended for use in situations in which numerous, similar systems will be created over time ...
>
> For a given application domain ... an analysis of this domain must be made and defined to Draco before it can be used to generate programs in the domain ...
>
> From the standpoint of software technology, Draco can be viewed as a system that provides two main functions: 1) the definition and implementation of languages of various types (properly viewed as specification languages) and 2) assistance in and partial automation of the task of refining a specification of a desired system (given in one of the languages known to Draco) into another language, presumably more concrete or executable, (also known to Draco). Draco provides assistance in optimizing the programs produced, managing the libraries of languages and their implementations, and performing other housekeeping details. [FRE87]

Draco has provided much insight into many related areas of computer science and software engineering. The prototype is being applied to realistic examples so that objective comparisons and conclusions may be made. Other related investigations are also continuing. One conclusion which has been reinforced is that "domain analysis and design is *very hard*." [NEI84] Most of the Draco domains which have been built so far are relatively low-level, pertaining to the Draco infrastructure (e.g., parser generation, transformation library construction, pretty printer generation). It remains to be seen whether the Draco approach can be applied to larger and more general-purpose domains.

3.1.2.4 ASLs and the SSAGS

Unisys has been researching the implications of reuse on the software lifecycle, and in particular, has been developing a particular technical approach to reuse based on an attribute grammar-based generative capability.

[SIMSS] In effect, they have combined the programming language and narrow domain approaches to automatic programming:

> At Unisys, our technical approach to software reusability is founded on the use of *application-specific languages*, or ASLs—very high-level non-procedural specification languages that employ syntax and terminology suitable for a specific, narrow-band domain of application. We have produced a number of ASL processors for diverse domains such as computer system configuration and message format translation/validation. These processors translate specifications written in an ASL to efficient, maintainable high-level language code.
>
> Generation of high-level code from ASL specifications has proved a better approach to software reuse than libraries of reusable components in these domains, due to many factors. Besides the productivity gain obtained from use of a concise, declarative description, ASLs permit the generated code to be driven by the specification in a far more complex way than is permitted by parameterization in a general-purpose programming language (even Ada, with features such as default parameters designed to be particularly supportive of reuse). Semantic checks peculiar to the application domain can be incorporated into the structure of the ASL; moreover, by embedding domain-specific algorithms and expertise in the code generation component, generated high-level source code can be of comparable efficiency to a hand-written implementation. This makes ASLs potentially more viable in the near term than general-purpose 'automatic programming' systems; such systems tend to employ highly formal, abstract syntax, and cannot draw as readily on efficient application-specific code generation techniques. [SIM88]

In addition, Unisys has developed an operational system—the Syntax and Semantics Analysis and Generation System (SSAGS)—which "enables many portions of an ASL language processor to be generated from a formal specification of the ASL." [SIM88] With this system, they have "significantly reduced the effort required to develop, maintain, and retarget ASL translators." [SIM88]

The Unisys researchers have launched a two pronged attack: 1) manually developing domain languages (ASLs) and their processors, and 2) automating (via SSAGS) the development of the domain language processors. By making use of the domain language technology from their first effort, they are proceeding to develop SSAGS through a bootstrapping process. This strategy is an effort to prepare for the future transfer of their technology into other domains. Their success is currently limited to narrow, low-level domains. This seems to be a recurring phenomenon. However, the SSAGS is mostly domain-independent, and their overall approach is well founded in known computer programming language technology.

3.1.2.5 Frame-Based Software Engineering

Netron, Inc., based in Ontario, Canada, is advocating a frame-based approach to software reuse. [BAS87] This technology—not to be confused with AI frames—is firmly based upon the concept of adaptable, reusable components. Bassett provides a very eloquent discussion of reuse engineering via automated adaptation and composition:

> Frames ... provide a basis for a rigorous software-engineering discipline. Generally speaking, a frame is any fixed theme plus the means to accommodate unforeseen variations on that theme ...
>
> [C]onsider manufacturing: In about 20 minutes you can order a new car that is as unique as your fingerprints. In principle, every car on the assembly line can be one of a kind. How can they be made in high volume and quality at reasonable cost? When you tour an automobile plant, the first thing you notice is that every car on the line looks the same. Of course, you are looking at the frames. The unique results are obtained from the combinatorial explosion of options that can be bolted, sprayed, and welded onto the frame. And because the frame is engineered for such options, hundreds of millions of dollars can be invested in automatic assembly equipment such as robot welders.
>
> Programs are variations on themes that recur again and again. A software-engineering frame is a model solution to a class of related programming problems containing predefined engineering change points. [BAS87]

Netron has developed a frame formalization which facilitates the use of language-independent frames. The frame syntax is described with a modified Backus-Naur Form (BNF) grammar, and the semantics are said to be a form of macros. Netron reports extensive experience with COBOL applications, and some with LISP and Ada. Netron's parent company has apparently developed and installed 20 million lines of custom COBOL, forming numerous application programs, "composed from 30 input/output frames, 38 application-oriented frames, about 400 data-view frames (defining more than 10,000 data fields), screen and report frame generators, and one custom specification frame per program." [BAS87] Frame-based software engineering, as described by Bassett, hinges on the *specification frames*:

> Of pivotal importance are the specification frames. Each program corresponds to one specification frame, which is the root of a frame hierarchy. The specification frame controls the hierarchy's composition of the program and stores all its custom aspects. Specification frames typically contain less than 10 percent of a program's source code—and specification frames are the only Cobol frames for which application developers are ever responsible.
>
> Specification frames are created from template specification frames. A template defines a default frame composition for an application domain and provides all the important options, with explanations to guide the choices. The two-dimensional hierarchy of options is flattened into a linear list. The

> application engineer, unaware of the underlying tree structure, customizes a renamed copy of the template. The resulting specification frame, with the underlying frames, is processed automatically (including a compile and link) to produce an executable load module. [BAS87]

Applications within the Management Information System (MIS) domain—typically implemented in COBOL—have long been a catalyst for advanced reuse technology, due to their relative stability and high degree of commonality. [SIM88] Netron's frame-based technology is yet another case in point. While some experience in Ada has been reported, only an internal 'reusability analysis' technical report is referenced. [BAS87] Although the evidence provided is very attractive, it is completely unknown if this technology can be applied effectively outside of the MIS/COBOL arena.

3.1.2.6 Meld

Meld is an object-oriented, declarative language designed to facilitate the composition of software systems from reusable building blocks. [KAI87] Meld is an outgrowth of work done on the Gandalf project at Carnegie Mellon University. Meld has two essential aspects that set it apart from other object-oriented languages, *features* and *action equations*:

> **Features.** Features are reusable building blocks. Like Ada packages, their interfaces are separate from the implementation. A feature bundles a collection of interrelated classes in its implementation. The interface exports a subset of these classes and a subset of their methods. In effect, a feature is a reusable unit larger than a subroutine, on approximately the same scale as an Ada package, that permits the reuse of the glue among subroutines (methods) and, in fact, among abstract data types (classes) ...
>
> **Action equations.** Unlike classes, however, features can combine, as well as augment and replace, inherited methods. This is accomplished by writing methods as action equations. Action equations should not be confused with mathematical equations. They were developed to extend attribute grammars for semantics processing of programming environments.
>
> Action equations define (1) the relationships that must hold among objects and among parts of objects and (2) the dynamic interaction among objects and between objects and external agents (such as users, the operating system, and utilities). [KAI87]

Action equations consist of methods—in the object-oriented sense—and unidirectional constraints. In terms of automatic programming, Meld follows the programming language and transformational approaches. The Meld implementation translates the (language-independent) features and action equations into a conventional programming language, and supports the execution of such systems with a special run-time environment. The translation algorithms were derived from previous work on software

generation. The run-time environment supports object and external agent primitives, and provides a constraint satisfaction system. The Meld system has been evolving over several years, and is currently being bootstrapped from the previous implementation. It is reported that the previous system "has been used to implement demonstration-quality, multiple-user programming environments for small subsets of Modula-2 and Ada." [KAI87]

This research is another example of applying constraint technology to the object-oriented paradigm; evidently a powerful combination. As with Gist, the resulting declarative nature of the system has led to bootstrapping and perpetual prototyping. As far as capabilities, Meld provides a means for creating reusable components which are clusters of *objects*. This is an interesting capability and may prove to be of value for object-oriented reuse libraries. Meld's action equations also provide a more powerful composition mechanism than object-oriented *methods*.

3.1.2.7 CAMP

The CAMP component constructors were analyzed in detail in Section 2. From the automatic programming perspective, the CAMP effort obviously followed the narrow domain approach. In addition, the CAMP constructors can be viewed as an effort to extend the capabilities of the Ada programming language; following the programming language approach.

3.1.2.8 Prototype System Description Language (PSDL)

Researchers at the Naval Postgraduate School have been developing a Prototype System Description Language (PSDL) for investigating knowledge-based support for rapid software prototyping. [LUQ88] Luqi describes the unique aspects of their research through a comparison with the Programmer's Apprentice KBEmacs prototype:

> Our approach differs from other knowledge-based approaches to program construction by (1) the scale of its knowledge base, and (2) its computer-aided retrieval of reusable components based on specifications. The Programmer's Apprentice project aims at speeding up the programming process via a library of reusable components. Reusable components implemented in the KBEmacs version of the Programmer's Apprentice—components known as "clichés"—represent algorithm fragments rather than complete modules. Using algorithm fragments, KBEmacs focuses on supporting the assembly of a module's implementation rather than assembling systems from complete modules; the scale of examples reported is a few hundred lines of code. We aim to produce software system prototypes larger by several orders of magnitude, and seek to avoid considering the internal structure of reusable components. [LUQ88]

A large portion of the PSDL research is focused on retrieval based on specification. We will not address these issues here. However, this system combines reusable component retrieval and adaptation in a very interesting manner which attempts to alleviate the problems caused by finite and rigid component libraries:

> We can provide transformations that adapt or combine components explicitly stored in the software base [the knowledge-based equivalent of a component library] to accommodate a class of small variations, thereby alleviating this problem. Systems performing such transformations as part of the retrieval process have a much better chance of successfully retrieving a specified component from the software base than do systems that can only return components explicitly stored in the software base. This capability also reduces the need for designers to manually adapt reusable components after retrieval ... [LUQ88]

This effort seems to be directed at solving the perceived problems of the Programmer's Apprentice cliché libraries. It is reported to be in the very early stages of development. The concepts being explored are indeed very attractive. One negative aspect of the project is that they have restricted their research to the dataflow and functional decomposition models of software composition.

3.1.2.9 Software Templates

Researchers at the Oregon Graduate Center have developed an approach to reusable components which is based on the separation of algorithms—embodied in *software templates*—and (primitive and abstract) data type implementations. [VOL85] Software templates "are defined over values of abstract data types whose implementations are specified separately and catalogued. When a template's data types are bound to catalogued implementations, the template is automatically translated into a component tailored to the chosen implementations ..." [VOL85] Biggerstaff and Richter describe this research as:

> An interesting approach to the prob.ems of composition ... By separately choosing an algorithm and its implementation decisions for the data types used, the system generates a customized implementation of the algorithm, often producing a significant code expansion. It remains to be seen how far this work can be pushed. [BIG87]

Thusfar, a prototype system has been developed, with a small library of implementations (written in 'C' and Pascal), and an enhanced version of the prototype is reportedly under development.

3.1.2.10 Software Construction Mechanisms

From the above discussion, it is evident that numerous research projects are exploring new and innovative methods of software construction. This research will eventually lead to enhanced state-of-the-practice programming environments. Relatively speaking, today's programming environments provide a number of low-level construction mechanisms.

For example, object-oriented systems provide **inheritance, specialization** and **generalization**. Specialization and generalization are manual construction mechanisms. The combination of inheritance and specialization are especially effective software construction mechanisms. Using this technique, new capabilities can be developed incrementally and with ease; reusing previously developed classes and building from them. Generalization is the creation of a new class from two or more previously developed classes. Generalization actually provides an indirect construction benefit: two or more classes are merged together in this manner when the resulting class is expected to provide a better source for inheritance, thus promoting reuse and future construction.

Thusfar, we have expounded the object-oriented mechanisms in terms of composition and construction. These same mechanisms also support adaptation. The inheritance mechanism facilitates adding, deleting, and restructuring classes within an inheritance hierarchy. In addition, dynamic binding allows such restructuring without necessarily having to modify the effected classes. When examining the suite of object-oriented mechanisms—in terms of composition, adaptation and construction—it becomes obvious why this software paradigm is so powerful.

Generics—as provided by Clu and Ada—are primarily adaptation mechanisms. However, the associated **instantiation** capabilities can be considered construction mechanisms. With a predefined generic component, this mechanism facilitates the automatic construction of executable software components.

Macros provide a similar but less strict form of construction. As with generics, there are two sides to macro facilities: macro definition and **macro expansion**. Macros can be used as adaptation mechanisms, but they are primarily used as construction mechanisms. LISP macros are especially powerful. A LISP macro "is given a piece of program and computes a new piece of program from it. Thus MACROs exploit the fact that LISP programs and data have the same form." [WIN81]

3.2 Composition and Construction of Ada Components

Software construction has long been compared to computer hardware fabrication which consists of printed circuit boards and integrated circuits (chips). This analogy has often been criticized because of the inherent differences in hardware and software, but it is useful nonetheless. In both cases, large complex structures are constructed, in part, by composing smaller well-defined components.

In the case of hardware, these components have become so well-defined that they are abundantly available commercially. Hardware engineers routinely browse through thick catalogs of components and are able to pick and choose among components which possess some desired specifications. While this has become standard hardware activity, *software reuse* is merely an emerging technology.

Ada provides a wealth of fundamental programming mechanisms which imply a coherent composition model. This includes procedures, functions, tasks, exceptions, packages, renaming, overloading and generics. However, as with other high level languages, this model of software composition has its limitations. Table 3–1 illuminates these limitations by comparing increasingly powerful Ada construction techniques with the corresponding hardware analogy.

Table 3–1 Ada Construction Techniques

Ada Construction Technique	Hardware Analogy	Adaptation & Reuse
Customized — program control structure and software components are custom built for each application	chips and board are custom built for each application	systems are built using the facilities provided by the currently available formalized level of abstraction (e.g., Ada, hardware fabrication tools)
Bottom Up — program control structure is built around existing software components	custom board is built to accommodate commercial or otherwise available chips	a set of building blocks may be available which provide complete or at least substantial coverage of a given domain
Top Down — existing program control structure drives the adaptation, selection, and/or creation of software components	semi-populated commercial board is completed with the selection and/or configuration of commercial chips	sets of similar components—providing coverage of specific but narrow domains—may be available for selection and/or parameterization
Generics — abstract, parameterized program control structures and software components are utilized as much as possible in the construction of a specific application	commercial board (populated) is tailored to a specific application via DIP switches (e.g., port selection, timer configuration)	components which have been designed for maximum adaptability (as facilitated by the current level of abstraction) are available for the construction (through adaptation and composition) of specific applications

Table 3–1 also analyzes the level of adaptability and reusability which is accomplished with each construction technique. In fact, the *power* of these construction techniques can be mostly contributed to the level of adaptability provided by the corresponding components. Table 3–2 explores the increasing levels of adaptability and reusability of Ada software components.

Table 3–2 Adaptability and Reusability of Ada Components

Level of Adaptability	Level of Reusability
singular (customized) component	use as-is if directly applicable; modify if partially applicable; or don't use
multiple components	select from finite set of similar components Ada components may provide multiple bodies for a given specification and/or multiple specification and body pairs
subprogram parameters	provides dynamic tailorability (within limits of data types)
generic parameters	provides static tailorability; can also be thought of as providing an infinite selection of similar components
construction (i.e., requiring specialized automation)	provides adaptation/abstraction capabilities beyond those of the current programming language

Table 3–2 provides a simplified view of the adaptability and reusability of Ada software components. However, it is enlightening to view the increasing levels of adaptability in relation to the underlying programming language capabilities. The Ada programming language—in particular the Ada generic facility—does not provide reuse engineers with a sufficient adaptation capability. The CAMP project provides evidence to support this. In order to achieve the desired level of adaptability, conventions were adopted and constructors were developed. *Construction (or generation) is the automated means to achieve a level of adaptability not available using a base programming language.*

3.3 The Evolution of Computer Programming Languages

We are evolving towards a higher level programming language. What happened on the CAMP project is a well-documented snapshot of this evolution. In each case, the CAMP constructors provide a level of programming which is more powerful than that of the Ada programming language. For example, the CAMP constructor dialogs are actually a form of computer programming language, providing a higher level, declarative, and interactive programming capability. The problem with the CAMP

constructors is that they are very specific, narrow, and ad-hoc enhancements of the Ada programming language. Nonetheless, they do provide programming capabilities beyond those of Ada.

This is a clear progression towards a higher level of abstraction which has happened time and again. Figure 3–2 illustrates the cyclic nature of this evolutionary process.

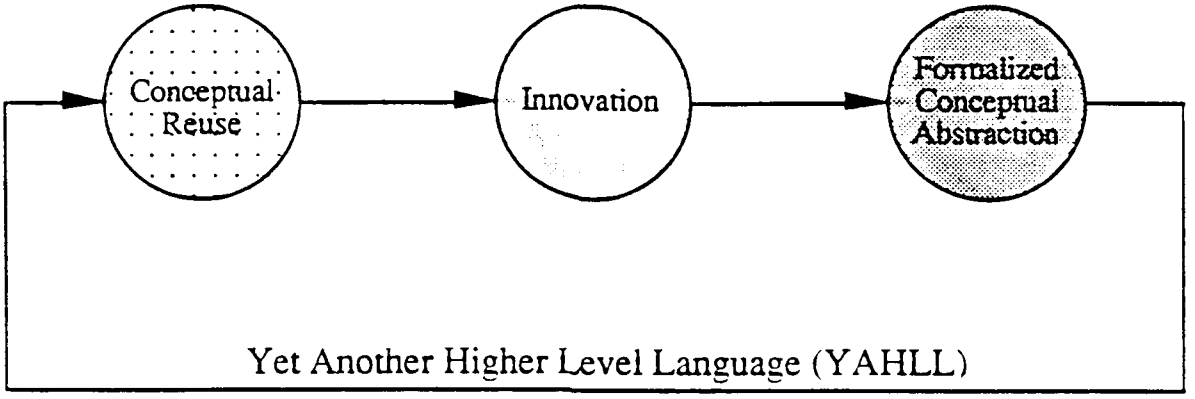


Figure 3–2 Technological Evolution

In terms of computer programming, once the reuse of a concept is formalized/standardized, it is eventually incorporated in a programming language. The concept may be incorporated as an extension, a modification, or as the basis of a new, evolutionary language. CAMP started with specific constructors and is now (CAMP–3) moving towards more generalized approaches. The constructors provide the programmer with a higher level of abstraction in which to work with. In turn, this increases one's expressive and reasoning power, leading to more compact, understandable, and maintainable programs. The net result is increased software productivity. Figure 3–3 provides a historical view of the evolution of computer programming languages.

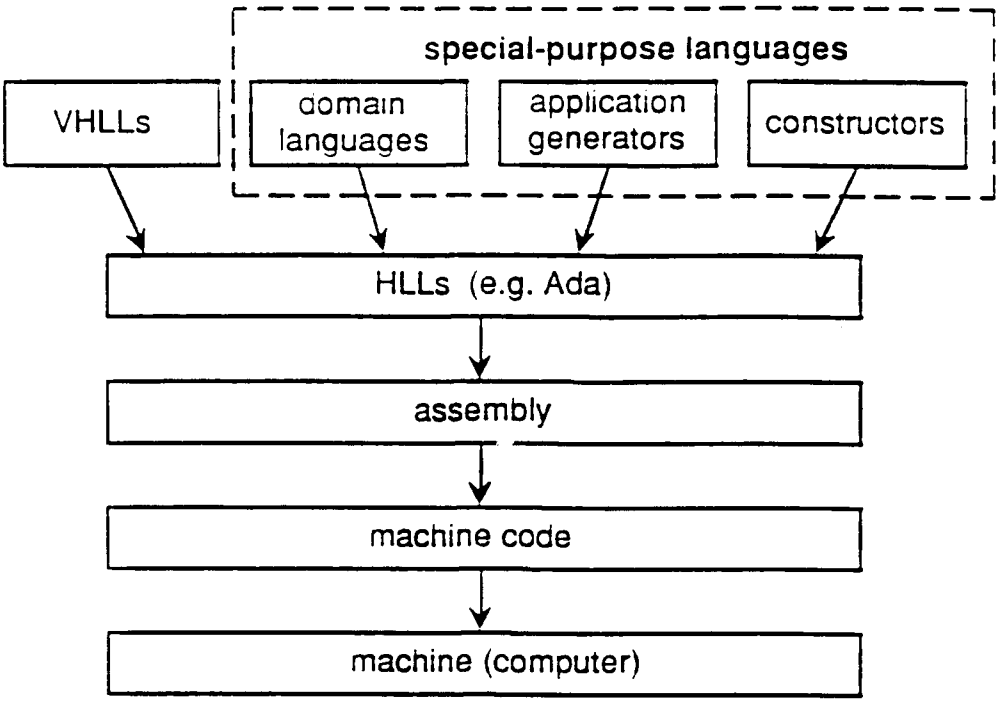


Figure 3–3 Evolution of Computer Programming Languages

Over the years, we have progressed from machine code computer programming through high level computer programming languages such as Ada. At this point, it is unknown what the next generation, i.e., very high level language (VHLL), general-purpose computer programming language will be; technological evolution tends to advance at a snail's pace. Of course, this is our view, since we are in the midst of it. In actuality, the technological evolution of computer hardware and software has been moving forward at warp speed in comparison to other technologies—past and present. Currently, domain languages, application generators, and constructors (i.e., special-purpose languages) are serving to push the frontier of general-purpose computer programming languages. In this sense, constructors can be viewed as *evolutionary vehicles*, as shown in Figure 3–4.

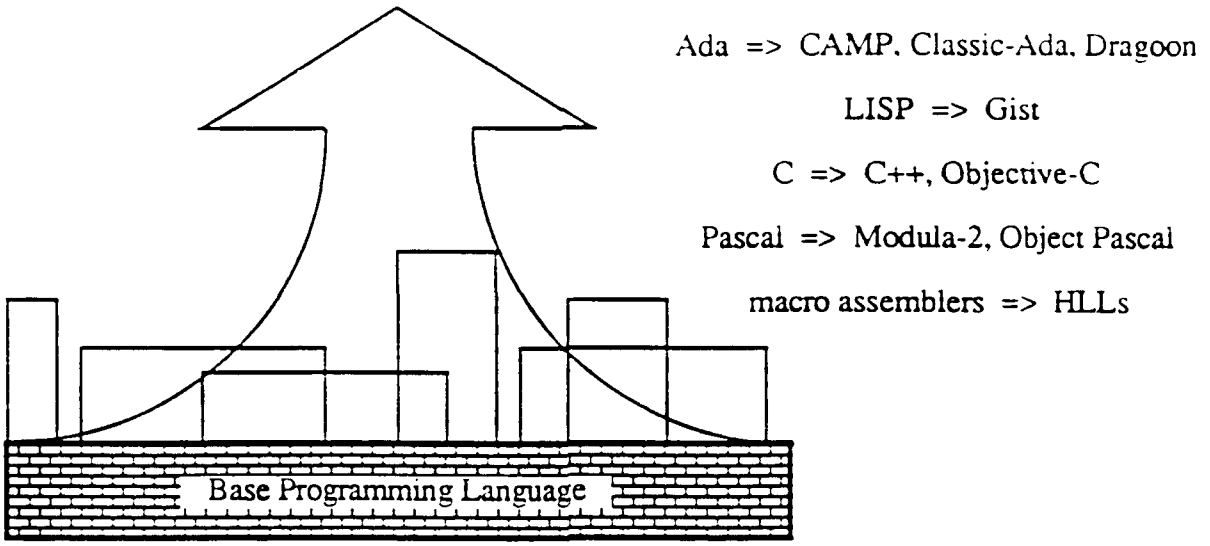


Figure 3–4 Constructors are Evolutionary Vehicles

There are several examples of this phenomenon, from the structured programming macro assemblers of yesteryear to the more recent Ada "constructors": Programmer's Apprentice KBEmacs and clichés, CAMP constructors, Classic-Ada™, and Dragoon. Classic-Ada is a CASE tool developed by SPS that supports object-oriented design and implementation with Ada. Dragoon is a similar product being developed in Europe, but is not yet commercially available. Classic-Ada provides extensions which support the object-oriented programming mechanisms of (single) inheritance and dynamic binding. The Classic-Ada tools generate highly portable, DoD standard Ada code.

Viewing constructors in this new light, we may now revisit the CAMP project for a deeper understanding of the CAMP constructors. First, we will take a fresh look at the CAMP subsystem composition constructors. These constructors primarily served to provide assistance in the composition of existing CAMP parts. The backbone of these constructors is the Semi-Abstract method (i.e., the convention of *default* generic subprogram parameters) which was used to develop the underlying reusable parts. This is an example of how reusable components, in order to actually reach a significant level of reusability, are becoming increasingly adaptable.

But what happens when we reach the next level of computer programming language? Will this new computing power make reusable components obsolete? The answer is no. We believe that there will always be some form of reusable software components. Currently, the technology of reusable, adaptable components is moving faster than that of programming

languages. Reusable components are becoming increasingly adaptable, and are steadily being applied to earlier software development phases—up through domain analysis. *As we repeatedly develop more and more advanced software components, we will correspondingly need to address more advanced adaptation and composition of these components.* In this sense, the research and development of adaptable and reusable components is fueling the evolution of programming languages. The CAMP subsystem composition constructors are an example of this phenomenon.

Secondly, we will take another look at the CAMP component generation constructors. These constructors vary greatly in detail. However, each provides an enhanced programming capability over the Ada programming language. For example, the Data Type constructor provides a general-purpose capability which offers a very limited—but distinct—increase in programming power. On the other hand, the Finite State Machine constructor provides a significant increase in programming capability, but is much less applicable in scope. Each of the CAMP component generation constructors was driven by situations in which Ada alone could not capture some recurring programming concept, although "most software engineers have a good idea of how ... [the component] can be implemented." [MCN88]

But what happens when we reach the next level of computer programming language? Will this new computing power make component generation obsolete? Again, the answer is no. Computer technology is forever being applied to more challenging problems. What today is regarded as a straightforward application of computing technology was unthought of just a few years ago. By the same token, programming concepts which are well-understood and considered tedious today (e.g., finite state machines) will someday be abstracted away into more powerful and yet still general-purpose programming languages. Then, after applying this new programming power to even more challenging problems, the need for component generation will arise once again. *As we repeatedly apply programming languages to more and more challenging problems, we will correspondingly need to address more advanced software construction capabilities.* The CAMP component generation constructors are an example of this phenomenon.

Having gone full circle, beginning and ending with the CAMP constructors, we are now ready to address the initial task: generalized construction approaches. But first, we must reconsider just what constitutes a **constructor**. Ironically, this most basic of questions presented a difficult challenge throughout this research effort. Figure 3–5 provides a graphic illustration of our new outlook on constructors.

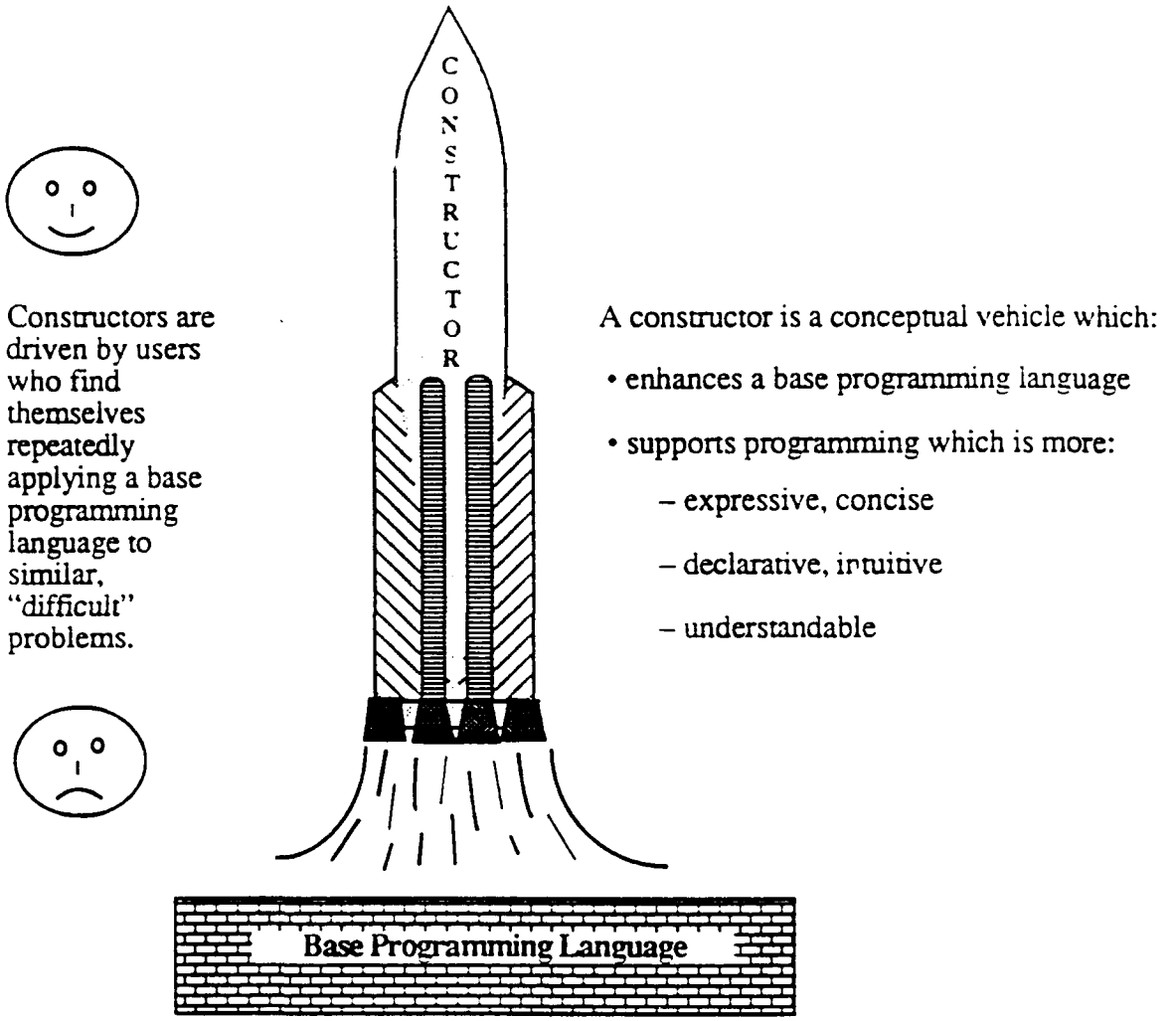


Figure 3–5 What Constitutes a Constructor ???

We can answer this question by examining the constructor from the perspectives of the reuser and the developer. Ideally, the reuser is a domain expert who is not required to understand the component's design, implementation, or even the implementation language. The reuser only needs to know the external behavior of the component, in terms of the domain. The reuser considers the constructor to be an adaptable, reusable component (of varying size and complexity). In order to use the component, the domain expert simply supplies the domain-level specification information necessary for the component's adaptation to a particular application.

On the other hand, the developer of a constructor is a software engineer. The developer views the constructor as a program, or software tool, which

generates a customized software component. The software component it is responsible for generating has a well understood design and implementation. However, the component must accommodate several domain related variances. These variances cannot be captured using the implementation language alone. The goal of the constructor is to eliminate the tedious manual work of constructing the component for use in specific applications. The constructor is to input the component's customization data and generate the appropriate source code.

We now have a better understanding of constructors and the issues concerning their generalization—programming languages and their evolution, software adaptation & reuse, and the fundamental programming mechanisms: *composition, adaptation,* and *construction.* With few exceptions, the CAMP constructors (represented by the left half of Figure 3–6) were designed and implemented as individual entities; without cohesion. As a result, *generalizing the CAMP constructors would do nothing more than ease the burden of creating similar constructors.* For the CAMP project—faced with the maintenance and evolution of the AMPEE system and Parts Catalog—this may be a worthwhile effort. The CAMP constructors have the potential to significantly increase software productivity and reliability, but only within a very limited scope. The CAMP constructors should be viewed as a set of ad-hoc solutions to a set of very specific software development "problems."

Moreover, the basic notion of generalized *constructors*—even outside of CAMP—does not address the real issues at hand. From the perspective of computer programming languages, constructors are evolutionary vehicles. Constructors are primarily short-term solutions; eventually to be replaced by more systematic, integrated and/or general-purpose programming capabilities. Merely by their existence, constructors are admittedly separate entities from their underlying programming languages. From the standpoint of integrated, seamless programming environments, this is an obvious disadvantage.

Thus, generalizing constructors themselves only solves part of the problem. Generalized constructors would be more cohesive and integrated, but they would still be separate entities, and they would still be separate from the base programming language. A good example of this phenomenon is DARTS (see Section 2.2.1.4 and [MCN86b]). The DARTS technology includes an extendable specification language called AXE. Once extended to support a particular application domain, AXE statements are embedded within the source code of existing components. Utilizing these genericized components, the DARTS system is able to generate customized components from domain-level specifications.

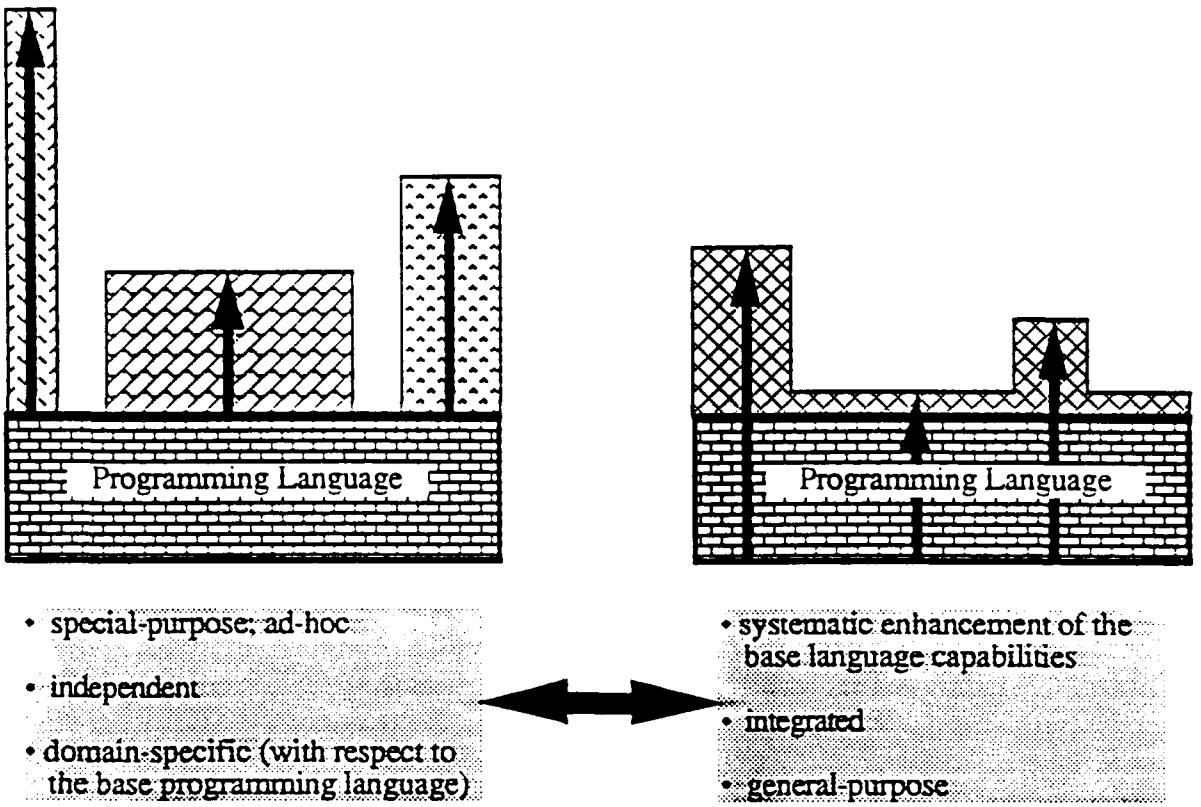


Figure 3–6 Constructor Implementation Spectrum

DARTS is a generalized constructor; a single system which is capable of generating numerous components. However, the generalization is not an enhancement of the base programming language. In fact, AXE statements can be embedded in any programming language. At first, this seems like a desirable characteristic for generalized construction approaches. DARTS' developer, General Dynamics, even planned to make the technology commercially available. This never happened. After some use internally, General Dynamics is now using only pieces of the DARTS technology (not including the AXE language). We believe that one of the reasons for DARTS demise is the fact that the DARTS approach, though generalized, is not an integrated enhancement of base programming language capabilities. DARTS users have to combine the syntax and semantics of two completely separate languages into a single software component. Embedding foreign constructs within the nooks and crannies of software components is equivalent to crafting individual component constructors. Any generalized construction approach which utilizes programming formalisms detached from the base programming language (e.g., DARTS, constructor-constructors) will result in the development of reusable components which suffer in terms of

understandability and maintainability, and will be entirely dependent upon the associated special-purpose software system.

Our new understanding of constructors has led us to the conclusion that truly generalized constructors are implemented such that the fundamental programming mechanisms of the base programming language are enhanced, illustrated by the right half of Figure 3–6. In this manner, constructors can form a more seamless integration with the programming language, while also promoting its eventual enhancement and/or evolution.

We believe that constructors are vital to the evolution of programming languages. We also believe that constructors can be valuable in the here-and-now. Especially in domains with a large degree of commonality, constructors have the potential to promote reuse and to significantly affect productivity and reliability. In fact, the following section presents a promising scenario of how the GCC may be integrated within future reuse environments.

4 Development Environment Integration

One of the objectives of this effort is to investigate the integration of the Generalized Constructor Capability with existing, commercial software development environments. Two kinds of development environments of primary importance are expert system development environments and software reuse environments; these are addressed in the following sections.

4.1 Expert System Development Tools

This section evaluates the suitability of using an expert system development environment to implement the GCC. In particular, we will recommend whether or not the Automated Reasoning Tool (ART), or a similar expert system tool, should be applied to the GCC.

Although CAMP Phase 1 concluded that expert system technology should be applied to the constructor problem, CAMP Phase 2 found that, for the most part, an expert system was not required. Despite the fact that the Automated Missile Parts Engineering Expert (AMPEE) system was implemented with ART, "[t]he Parts Identification subsystem is the only portion of the AMPEE to use ART for anything more than data structuring." [MCN88a] While ART provided a convenient prototyping environment, the functionality of AMPEE could be accomplished using other more conventional techniques. In fact, the Parts Composition System (PCS) has been re-engineered in Ada as part of the CAMP–3 project (see Section 2.3).

Although originally conceived as an expert system, AMPEE makes only limited use of ART functionality:

> It is used throughout the AMPEE system for data structuring (via the ART schema system [which includes inheritance]), and within the Parts Identification subsystem for consistency checking and interface control (via a small number of simple forward-chaining rules), and for display of the missile software hierarchy within the Missile Model Walkthrough function. ART provides many more features that are not used in the AMPEE system, such as backward chaining rules and the ability to explore alternative scenarios... [MCN88a]

The CAMP developers warn that the use of complex expert system development environments such as ART impose limitations on the final system, including portability, cost, and compatibility. (There are, however, fairly powerful shells and tools that are implemented in standard programming languages and run on conventional platforms.) They suggest that "before utilizing such a tool, it would be beneficial to determine which of

its features are likely to be needed, and determine if some or all of the needed features are available in a simpler and more portable package or language." [MCN88a]

However, expert system implementations of young and emerging technologies, such as an automated parts composition system, are capable of easily accommodating advances and evolution of the maturing technology. This was one of the reasons that expert system technology was originally recommended for AMPEE, and is a definite advantage of a (good) expert system implementation.

Even so, *we do not currently recommend the use of ART, or a similar expert system tool, for the development of the GCC.* Expert systems, as the name implies, are capable of modeling specialized problem-solving expertise. As evidenced by the lack of requirements specified during this effort, we have yet to develop a sufficient understanding of the expertise necessary to adapt reusable software components for specific applications. If, after developing the GCC requirements and/or a GCC specification, it becomes clear that some aspect(s) of the GCC are best suited for an expert system, then it would be appropriate to impose expert system implementation requirement(s).

On the other hand, the object-oriented paradigm offers an increasingly attractive means of achieving the same flexibility requirements. Our corporate experience leads us to believe that the object-oriented paradigm is well suited for the implementation of the GCC—as well as its specification and design. We believe that an object-oriented implementation would provide the needed adaptability and extensibility (as expounded upon in Section 3.1.2.10). It is for these same reasons (e.g., inheritance, dynamic binding) that prototypes and simulations are often implemented in object-oriented languages. In addition, the object-oriented paradigm provides an excellent framework for modeling and classifying reusable software components (more on this in the following section). Finally, specialization provides a very powerful adaptation mechanism for software components, even though we don't yet understand the specific kinds of adaptation which the GCC will need to support. In addition to all of this, the continued maturity of object-oriented programming languages and environments would ensure good productivity and quality for the GCC development effort. It should also be noted that there is a continued merging of the object-oriented paradigm with knowledge representation and reasoning systems. Thus, if aspects of the GCC do indeed lend themselves to an expert system implementation, it will be easy to integrate the object-oriented modeling with the expert system's reasoning capabilities.

4.2 Reuse Environments

Another aspect of this task is to investigate the implications of reuse environments in supporting the Generalized Constructor Capability. The GCC must necessarily be tightly integrated with the part definition and parts storage and retrieval capabilities provided by its reuse environment. Given the current lack and immaturity of the tools in this area, we will limit our investigation to using the Automated Reusable Component System (ARCS) as a platform for the GCC.

4.2.1 Automated Reusable Component System (ARCS)

ARCS is a Small Business Innovation Research (SBIR) program for which Software Productivity Solutions, Inc. (SPS) has recently been awarded a Phase II contract with CECOM (Contract No. DAAB07-89-C-B920). SPS is in the process of developing a production-quality reusable software component library management system with the following features:

- powerful information query and browsing
- multi-window user interface
- support for multiple user roles
- flexible classification scheme definition and management
- extensible component submittal and extraction
- **support for the integration of extraction/adaptation tools**
- usage auditing and reporting

The following sections provide a more detailed description of the ARCS library management system. In particular, the ARCS concept of external support tools and their integration and cooperation is presented. It should be noted that the following sections represent work-in-progress and as such are subject to change.

4.2.1.1 Reuse Library System (RLS)

The ARCS library management system is called the Reuse Library System (RLS). Figure 4–1 shows the Reuse Library System and its external interfaces.

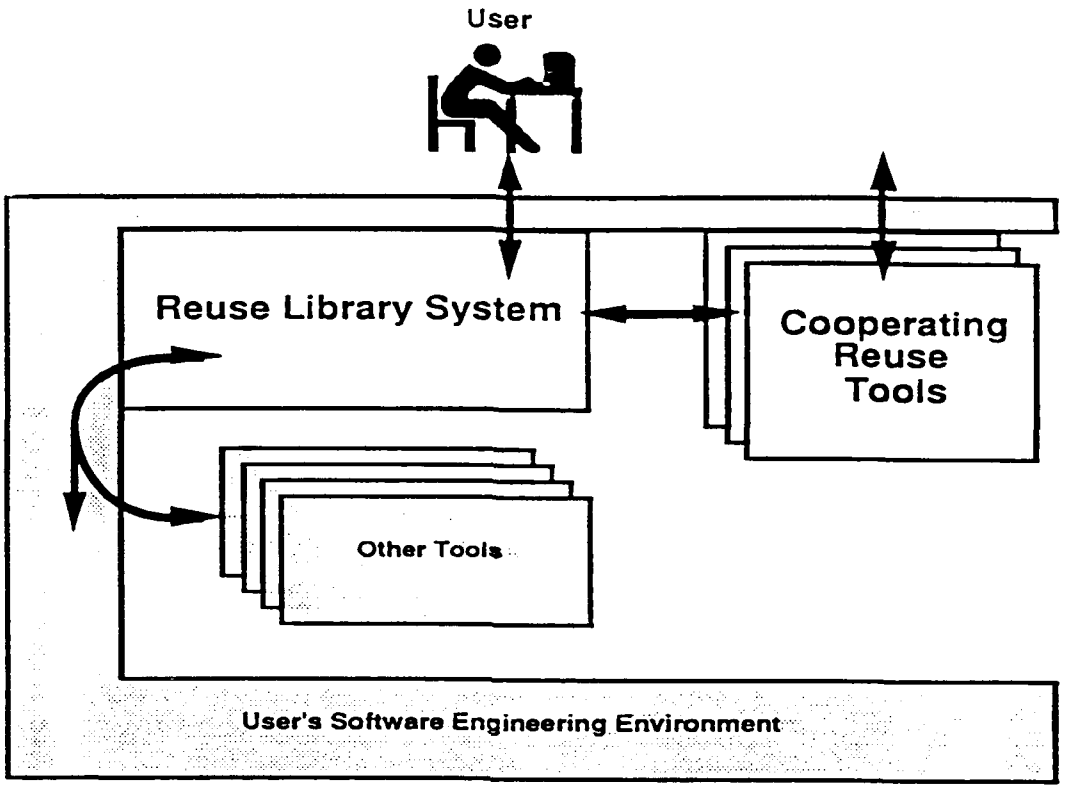


Figure 4–1 Reuse Library System External Interfaces

Users of the RLS have three primary roles: classifier, cataloguer, and searcher. Classifiers derive and maintain the component classification scheme. Cataloguers add components to the library. Searchers find and extract components from the library.

A distinguishing and innovative feature of the RLS is that it utilizes an object-oriented component classification scheme for the organization and management of the reusable components. The classification scheme is a component class hierarchy which specifies the kinds of components that can be described and searched for in the RLS. The classifier defines component classes and their associated attributes. Component classes inherit attributes from their superclasses. The following types of attributes may be defined:

- Single line of text — used for simple, short text values such as name, author, location, etc. Classifier can specify that values provided be unique.
- Multi-line text — used for large text values such as prose descriptions, source code, etc.
- Facet — controlled vocabulary (values must be one and only one of the approved terms defined for facet).

- Keywords — list of single line of text values, typically used for application keywords, project keywords, etc.
- Integer — for metrics-based criteria
- Fixed point — for metrics-based criteria
- Date
- Link — used to describe relationships between components, has specified cardinality.

Components are uniquely identified collections of attribute values that represent objects that have been catalogued and can be searched for, examined, extracted and reused. A component is an instance of a component class and is catalogued by providing values for the attributes which are defined by the component's class.

A **component set** is a collection of components produced as the result of a query. A **keep set**, on the other hand, is a collection of components which have been identified by the user for subsequent extraction.

The RLS supports user-defined layouts of components for cataloguing, browsing and searching purposes. A layout identifies specific attributes to be displayed/used and their general arrangement on the display. The searcher can display, file and print only those attributes of interest.

4.2.1.2 Checkout Tools (CTs)

As shown in Figure 4–1, there is a class of tools which have been identified in the RLS specification called Cooperating Reuse Tools. These tools support the reuse process, but are external to the RLS. They have the ability to communicate concurrently with the RLS, with each other, and also with the user.

The concept of Cooperating Reuse Tools is based upon a layered communication system composed of a low-level communication facility (e.g., TCP/IP), sockets, and an object exchanger. Sockets provide a means for concurrent communication amongst processes (i.e., tools). The object exchanger—part of the RLS—handles the exchange of encoded, typed objects to and from the RLS. All of these together allow the RLS user to interact with the RLS, as well as any Cooperating Reuse Tools, in a seamless, coordinated fashion.

Of particular interest are a special class of Cooperating Reuse Tools called Checkout Tools (CTs). The term *checkout* refers to the act of component extraction. Thus, Checkout Tools are responsible for the extraction of components from the searcher's keep set, including any necessary adaptation

(e.g., automated construction/generation). Indeed, it is envisioned that Checkout Tools will be largely responsible for the adaptation of components to suit the requirements specified by the searcher (i.e., the requirements imposed by the target application). Thus, Checkout Tools play a significant part in reuse by searchers who use the tools to extract and adapt components.

Checkout Tools are associated with the RLS component classes. Each component class may have at most one associated Checkout Tool. Checkout Tools are inherited—with overriding—through the component class hierarchy. Checkout Tools are registered by the Classifier. If no Checkout Tool is specified for a particular component class, then that component class inherits the Checkout Tool of its superclass. In this fashion, each Checkout Tool is responsible for the *set* of components which are instances of the corresponding component class(es).

With respect to component extraction, the following requirements have been allocated to the Reuse Library System, the Checkout Tools, the classifier, and the searcher:

- **Reuse Library System**
  - manage the keep set
- **Checkout Tools**
  - manage the extraction/adaptation of sets of components
- **Classifier**
  - register Checkout Tools (in accordance with the component classification scheme)
- **Searcher**
  - iterate through the keep set (selecting, browsing, and deleting/extracting components)

The RLS will insure the keep set to be persistent over multiple user sessions. Duplicates within the keep set will not be allowed.

Checkout Tools will manage any component extraction ordering and/or other component dependencies, as well as multiple session extractions. Dependencies will be managed by: 1) extracting a component and all of its dependents, and 2) placing component dependents in the user's keep set for their eventual extraction. Checkout Tools which engage in searcher dialogs will need to remember the answers to "common" questions (i.e., questions which pertain to the adaptation of more than one component).

The classifier will take the Checkout Tools into consideration when deriving/maintaining the classification scheme. The searcher will be responsible for iterating through the keep set. This allows the searcher to

keep intellectual control over the extraction process, choosing which components to extract and when to extract them.

Checkout Tools represent the Generalized Constructor Capability within the Reuse Library System. Therefore, our investigation of the integration of the GCC within reuse environments has focused upon the Checkout Tool. This effort has resulted in the development of a protocol defining the communication between the RLS and the Checkout Tools. This protocol is presented in the following section. The protocol is then followed by an operational scenario, providing an in-depth look at the integration of the Checkout Tool with the Reuse Library System.

4.2.1.3 Reuse Library System/Checkout Tool Protocol

Figure 4–2 specifies the Reuse Library System/Checkout Tool protocol. The protocol shown assumes a single user. The communication of user IDs may be needed in order to facilitate multiple users.

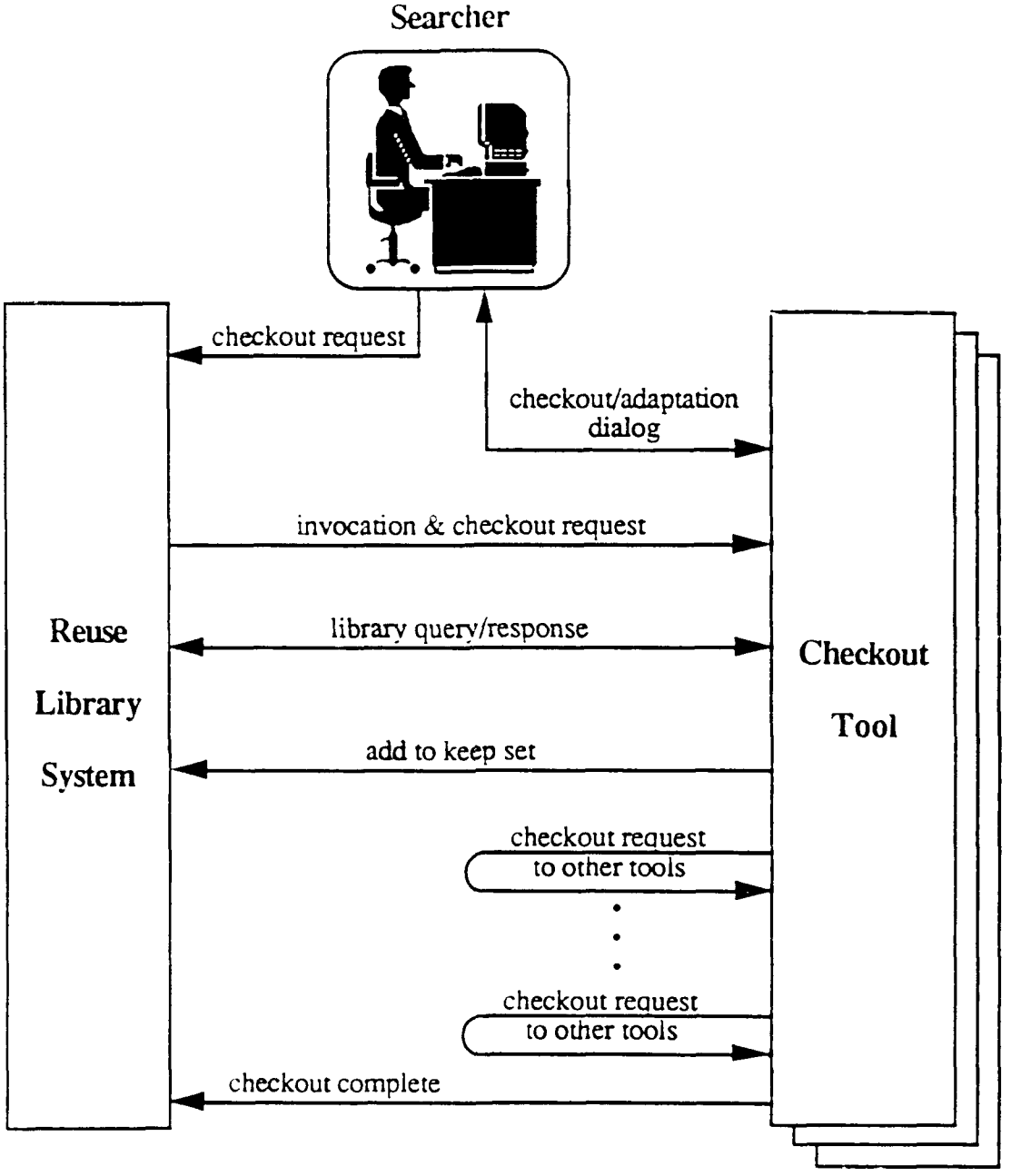


Figure 4–2 Reuse Library System/Checkout Tool Protocol

Figure 4–2 shows a searcher communicating with both the Reuse Library System and one or more Checkout Tools. On the user's display, this communication will take place in separate "windows." Depending on the look 'n feel of the Checkout Tool(s), the searcher may believe to be communicating with a single, coherent, reuse library system.

Typically, the searcher will select one or more components from the keep set and requests to check them out from the library. During the checkout process, the searcher may become involved in a dialog concerning the extraction and/or adaptation of certain components. For example, if a relatively time consuming extraction—especially of dependents—is about to be performed, then the searcher may be queried as to whether the extraction is to be performed now, or at another time.

Once a searcher has made a checkout request, the RLS will establish communication (via a socket) with the appropriate CT(s) and issue the corresponding checkout request(s). A set of component names will accompany this checkout request. From this point, the Checkout Tool is responsible for the checkout, and the RLS is passive until acknowledgement of checkout complete.

Checkout Tools may receive checkout requests from the RLS, or from other Checkout Tools. In the event that a CT needs to invoke another CT (an example of Cooperating Reuse Tools), a socket will be used to establish direct tool-to-tool communication.

During the act of checking-out components, the CT will most likely communicate with the RLS to request certain attributes of a component (e.g., source code, dependencies). As mentioned earlier, the CT may also enter a dialog with the searcher, soliciting adaptation requirements or other information. If a component needs to be added to the user's keep set (e.g., if the extraction of a dependent component has been deferred), then the CT may direct the RLS to do so. Upon completion of a checkout task, the CT will notify the RLS and provide a list of all the components, including subsequent dependencies, which have been checked-out.

The operational scenario of the next section will provide addition detail concerning the Reuse Library System/Checkout Tool protocol.

4.2.1.4 Operational Scenario

The following operational scenario provides a useful perspective from which to analyze and reason about the integration of "constructors" within the Reuse Library System's environment. The scenario is centered around a hypothetical Classic-Ada (see Section 3.3) component class, and its associated Checkout Tool.

The scenario is concerned with the creation of a simple user interface application as depicted by Figure 4-3. The application being constructed is a primitive demonstration of a user interface management system (UIMS) called Classic-Looks. Classic-Looks is a commercial product which SPS is currently developing. Classic-Looks is a collection of reusable components—

implemented mostly in Classic-Ada—for building object-oriented user interfaces. Classic-Looks includes high-level facilities for building and manipulating windows, menus, forms, various kinds of graphics, etc.

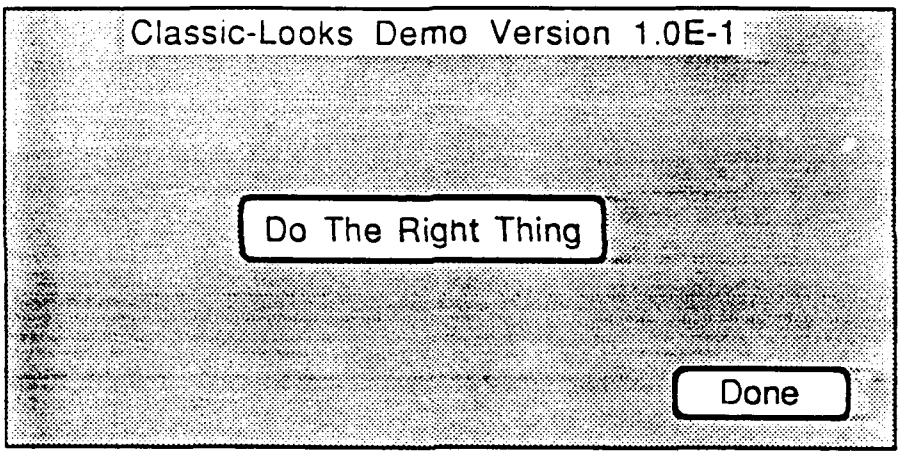


Figure 4–3 Operational Scenario: A Classic-Looks Demo

This particular version of the Classic-Looks demo displays a window with a text string at the top, and two buttons, each having a text string within them. The text string at the top is a title identifying the window as the Classic-Looks demo and providing the version number of the demo. The two buttons also have associated actions that are performed when they are "pressed." Other than their existence, the actual actions associated with the buttons are not relevant.

This scenario was chosen for several reasons. As mentioned earlier, we view Classic-Ada as a "constructor." As a software development tool, Classic-Ada provides capabilities which extend the base Ada programming language. Here at SPS, we have several software development activities which involve the use of Classic-Ada and Classic-Looks. In particular, and by no accident, numerous Classic-Looks components are frequently reused in the construction of user interfaces. Our engineers have found themselves repeatedly performing the same actions necessary to identify, retrieve, and adapt these components. As a result, we were able to devise an operational scenario which is complete (although relatively simple), detailed, and based on actual experience. While specific to the Reuse Library System and Classic-Ada, we believe the scenario is generally applicable to the integration of the GCC within reuse environments. In fact, the Checkout Tools themselves perform a GCC function (i.e., automated adaptation).

In reasoning about our experience with reusing user interface components, we hypothesized how the Reuse Library System (populated with Classic-Looks components) and a Classic-Ada Checkout Tool (CA CT) might automate this process. We believe that such a CA CT should, as a minimal

requirement, identify Classic-Ada related dependencies to other components. The operational scenario presented here explores this in detail.

For this scenario, assume that the Classic-Looks components have been catalogued within the RLS. Figure 4–4 illustrates the web of dependencies between the various top level components of Classic-Looks and other components necessary for this scenario. Figure 4–5 depicts the structure of the associated component classification scheme.

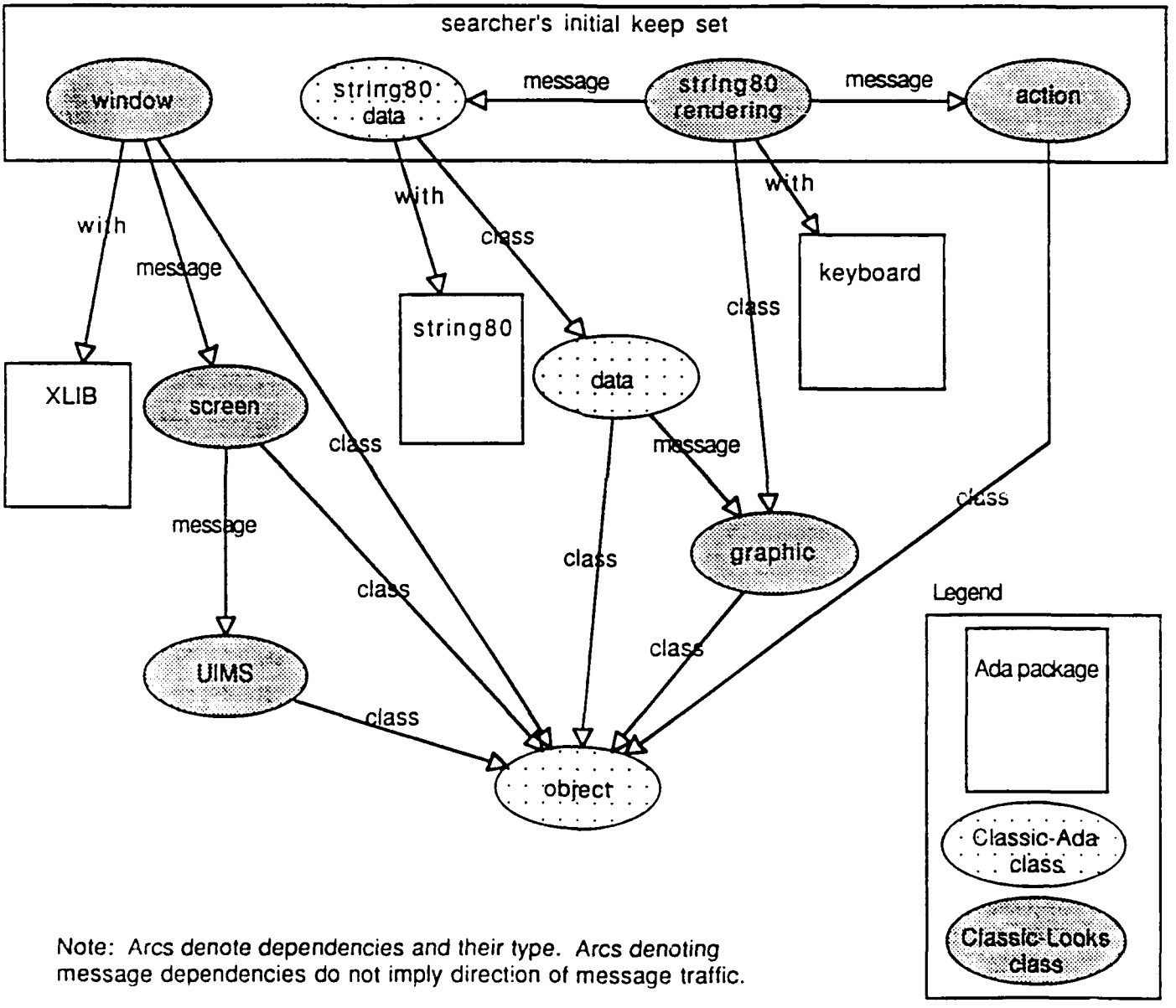


Figure 4–4 Component Dependencies

The components identified in Figure 4–4 are mostly self-explanatory. *XLIB* is an X Window System Ada library upon which Classic-Looks is built. *UIMS* is actually a misnomer. The UIMS component is an event dispatcher; it is not the root class of the entire Classic-Looks system, as the name implies. On the other hand, *object* is the root class of Classic-Ada, which explains why so many components are dependent upon it.

There are three kinds of dependencies which are identified in Figure 4–4: with, (super)class, and message. With dependencies are those found in Ada in which a component may be dependent upon one or more Ada packages explicitly through the with clause. Superclass dependencies are found in Classic-Ada components and are very similar to with dependencies. Superclass dependencies are explicit references to other Classic-Ada classes and represent the inheritance hierarchy. Message dependencies are implicit dependencies among Classic-Ada components. Unlike the with and superclass dependencies, they cannot be uniquely determined by examination of the source code. This scenario assumes that these dependencies are catalogued by the classifier.

In Figure 4–4, the message dependencies all represent implicit Classic-Looks protocol dependencies. For example, *window* is shown to have a message dependency upon *screen*. The reason behind this particular dependency is that *screen* (the Classic-Looks "engine") sends messages to *window*.

Although not represented in the dependency web of Figure 4–4, "separate" dependencies have also been identified as an additional type of dependency among Ada components.

The four components at the top of the Figure 4–4 represent the initial four components which a searcher would identify for the creation of the Classic-Looks demo described by Figure 4–3. From the Classic-Looks demo requirements, the searcher knows that the components *window*, *string80 data*, *string80 rendering*, and *action* are minimally required to implement this application. Obviously, the searcher must have some knowledge of Classic-Looks and Classic-Ada components. This knowledge may have been provided through the RLS. However, the searcher may not be aware of the various components which are required as a result of the dependency web shown in Figure 4–4.

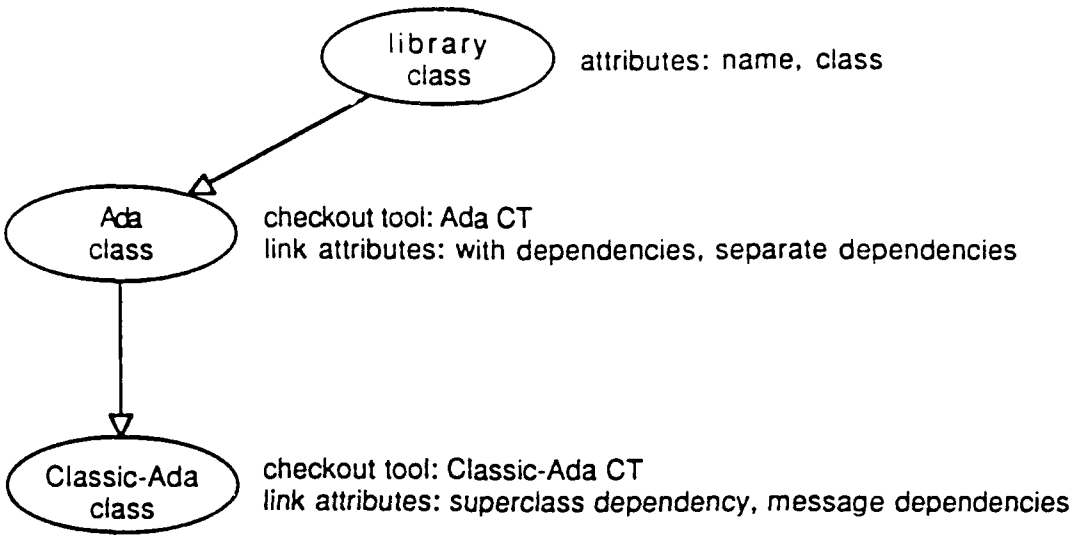


Figure 4–5 Library Classification Scheme

Figure 4–5 shows the library classification scheme which has been developed for this scenario. Through inheritance, each component class (and subsequently every component instance) will have a name attribute and a component class attribute. There are two Checkout Tools identified; one for Ada (Ada CT) and one for Classic-Ada (CA CT). Ada components are shown to have link attributes for with and separate dependencies. Classic-Ada components are shown to have additional link attributes for superclass and message dependencies.

As mentioned previously, most Classic-Looks components are implemented in Classic-Ada. Thus, these components are catalogued as instances of the Classic-Ada component class. The remaining Classic-Looks components are instances of the Ada component class. Note that neither of the associated Checkout Tools has any explicit knowledge of Classic-Looks.

The scenario, illustrated in Figures 4–6 and 4–7, show the interactions of the Reuse Library System, the Classic-Ada Checkout Tool, and the Ada Checkout Tool. The essence of the scenario is that a given CT checks-out all the components which it is responsible for, including dependencies. When it can no longer process any more components, it then sends messages to the appropriate CTs to process the remaining components.

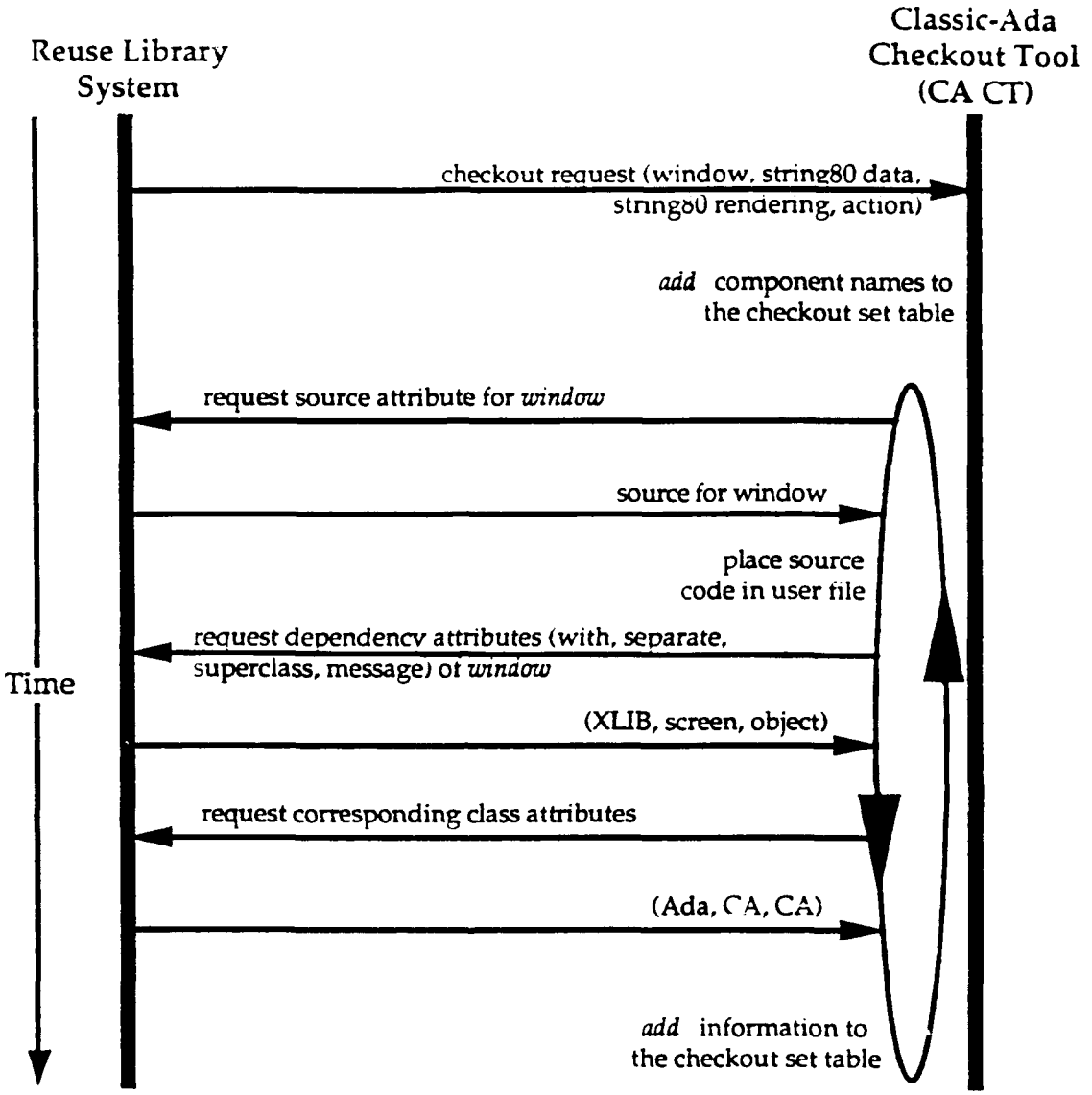


Figure 4–6 Operational Scenario

The scenario begins with a checkout request being sent to the CA CT for the initial four components of the searchers keep set, as depicted in Figure 4–4. The scenario assumes a checkout set table is maintained by the Checkout Tool, containing component names and corresponding component class names. After the component names are initially recorded, the component *window* is processed first, the remaining are then processed in order (as shown in the checkout request message).

The processing for *window*, and subsequent components, begins with the retrieval of the component's source code, which is placed in a buffer (a previously specified user file). In fact, this scenario covers nothing more than

the collection of the appropriate source code, including dependencies. More advanced checkout capabilities may include the creation of a translation/compilation command language file (i.e., a sequence of operating system commands), the selection of variant specs and/or bodies, and of course the adaptation or even generation of source code.

Once *window*'s source code has been extracted, the RLS is queried for *window*'s dependencies. For each of *window*'s dependencies, the RLS is then queried for the corresponding component class. This information is added (in the sense of sets) to the checkout set table. In this scenario, the checkout set table would now contain:

Component	Component Class
window	CA
string80 data	CA
string80 rendering	CA
action	CA
XLIB	Ada
screen	CA
object	CA

After *window* is processed, *string80 data*, *string80 rendering*, and *action* would be processed in the same manner. *XLIB* (a dependent of *window*) would not be processed by the Classic-Ada Checkout Tool. Continuing in this manner, the CA CT would then process *screen*, *object*, *data*, *graphic*, and finally *UIMS*. Eventually, all but *XLIB*, *string80*, and *keyboard* would be processed by the CA CT.

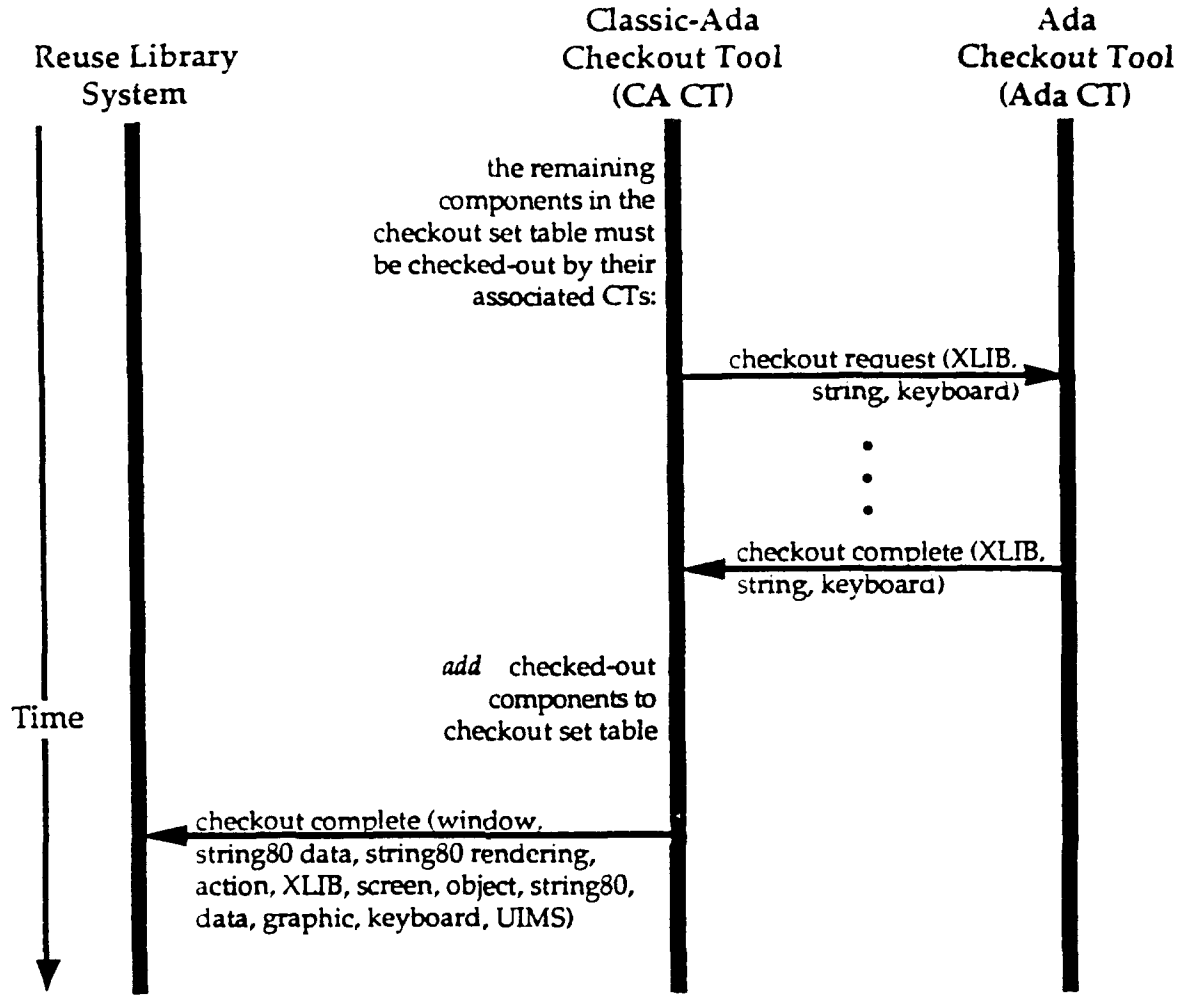


Figure 4–7 Operational Scenario (cont.)

At that point, the Classic-Ada CT would issue a checkout request to the Ada CT for the remaining components, as shown in Figure 4–7. Independently of the CA CT, the Ada CT would then perform the checkout of these three components, as well as any dependencies. It would then send a checkout complete message to the CA CT with a complete list of those components checkout-out. Likewise, the CA CT would *add* this list to its checkout set table, and return the complete set of checked-out components to the RLS. Notice the final set of checked-out components totals 12, compared to the initial request for 4 components.

4.2.2 The Integration of the GCC within Reuse Environments

Adaptation is prerequisite to reuse. The prime directive for Checkout Tools are to assist in the extraction and adaptation of reusable components

from the Reuse Library System. Assuming that a searcher has already identified a reusable component of interest, even if the component is a well encapsulated and cohesive unit of source code which needs no modification whatsoever, the mere act of extraction and incorporation into the target development is a form of adaptation. Furthermore, the required adaptation, no matter how simple or complex, must be performed *before* the component of interest is capable of being (re)used.

Identifying and managing inter-component dependencies is a reasonable adaptation requirement for Checkout Tools. For example, the operational scenario of the previous section demonstrated how a searcher requesting 4 components from the RLS resulted in the extraction of 12 components. The additional 8 components were identified though inter-component dependencies. Allocating this adaptation requirement to Checkout Tools allows searchers without detailed knowledge of the underlying component interdependencies to easily retrieve a complete set of components.

A more advanced adaptation requirement which might be imposed on a Checkout Tool might be to generate a command language file to be executed once a set of components have been extracted. The purpose of the command language file would be to complete the adaptation of the components, especially in terms of integrating the components into the target development environment. For example, the Classic-Ada Checkout Tool from the operational scenario could be required to generate a command language file for the Classic-Ada processing and Ada compilation of the checked-out components. In fact, our engineers believe this would be a significant aide in the adaptation of reusable Classic-Ada components. We believe that this type of adaptation is a feasible Checkout Tool requirement.

The Reuse Library System facilitates a very powerful Checkout Tool capability. This is due primarily to its object-oriented component classification scheme, and the concept of Cooperating Reuse Tools. Within the RLS environment, Checkout Tools:

- inherently support *sets* (i.e., classes) of related component instances,
- are capable of supporting multiple (hierarchically related) component classes,
- may just as easily perform "component construction" as "subsystem composition" functions, and
- have visibility to external component classes and instances, yet are cleanly separated and independent from them.

In the operational scenario, the Classic-Looks components were said to be instances of the Classic-Ada component class (see Figure 4–5). An alternative scenario might include a Classic-Looks component class and an associated Classic-Looks Checkout Tool. Presumably, a Classic-Looks CT would be

capable of advanced subsystem composition functions (in terms of the CAMP analysis). In fact, a sophisticated CT for the Classic-Looks components would be equivalent to what is commonly referred to as a user interface framework or development system. In more general terms, a "subsystem generator." Thus, a Checkout Tool governing a comprehensive set of components may house a subsystem/application generator, with the additional benefit of being one of many Cooperating Reuse Tools.

While a major criticism of the CAMP component constructors is their CAMP dependencies and domain specificity, it seems reasonable within the Reuse Library System to have Checkout Tools which are dependent upon and specific to components within a given component class (or subtree of classes). Indeed, it seems glorious, in comparison, to have a single tool which is capable of adapting whole sets of components, and without being dependent upon other tools or components. We believe that Checkout Tools will play a significant role in the usability of the Reuse Library System, and in the successful reuse of components catalogued within the library.

In summary, Checkout Tools represent the GCC within the Reuse Library System. This technology can be described as *object-oriented classification based adaptation on retrieval*. Adaptation on retrieval is not new. This form of adaptation is typified by the reuser's, or domain expert's, perspective. Once a reusable software component has been identified and selected for use, the reuse system adapts the component to the reuser's needs. In addition to this we add an object-oriented classification scheme; a flexible and extensible classification scheme in which a component class hierarchy specifies the kinds of components within the catalogue. We believe that this combination is the best integration of the GCC within reuse environments.

Furthermore, we believe that OO classification based adaptation on retrieval provides the opportunity for a much more aggressive, comprehensive, view of adaptation. For example, consider the following scenario:

- domain analyses which include:
  - a commonality study
  - an investigation of adaptation requirements, mechanisms, and implementations which enhance basic programming language capabilities
  - the development of a domain classification
- reuse environments which provide advanced adaptation support, including:
  - components organized into an object-oriented classification scheme

- components which are developed to take advantage of advanced adaptation capabilities, as provided for their class
- adaptation capabilities which are inherited and specialized throughout the classification hierarchy
- advanced adaptation capabilities for complex components, e.g., subsystems and generic architectures
- advanced inter-component adaptation, or synthesis

From this scenario, we get the feeling that the reuse environment has come to life, providing the reuser with a sophisticated collection of reusable components, subsystems, and even applications. We believe that OO classification based adaptation on retrieval, as presented, incorporates the essence of the GCC.

5 Conclusions and Recommendations

During the course of this STAS, we have analyzed several research and development projects (both from academia and industry), we have investigated expert system and reuse development environments, and we have even rediscovered one of our own commercial products. The journey has been one of investigation, of frustration (what is a constructor?), of learning, and of tracing pieces of Computer Science history and evolution. Drawing from all of this, we have come to several conclusions. They are sprinkled throughout the body of this report. Our most significant findings have been compiled into relevant categories and are presented below.

5.1 Summary of the Conclusions

Concerning the CAMP constructors:

- The CAMP component constructors are hard-coded; there are no *meta-parts* or *schematic parts* within the CAMP Parts Catalog.
- To the best of our knowledge, there is no general-purpose *Generic Instantiator Constructor.*
- The CAMP constructors generally fall into two classes: *subsystem composition* and *component generation.*
- The CAMP constructors have limited applicability outside of the armonics domain, are highly CAMP dependent, and exhibit low modifiability.

We believe that the two classes of component constructors identified within the CAMP analysis are due to the following fundamental truths:

- There will always be some form of reusable software "parts," driving the need for comparable **software adaptation and composition** facilities.
- The repeated application of a programming language to similar, "difficult" problems will always drive the need for advanced **software construction** facilities.

Concerning adaptation and its significance to reuse:

- Adaptation and its automation are key to software reuse.
- While difficult, separating adaptation requirements from adaptation mechanisms is essential to the specification and engineering of reusable software parts.

- The object-oriented paradigm provides a powerful suite of mechanisms which support composition, adaptation, and construction.

Concerning "constructors" and their generalization:

- Constructors can be described by any of the following:
  - reusable components which require special automation in order to adapt to specific applications
  - software programs which generate application-specific source code when provided with customization data
  - an automated means to achieve a level of abstraction or adaptability not available using a base programming language
  - a natural part of programming language evolution (e.g., language extensions such as Classic-Ada)
  - special-purpose VHLLs (e.g., domain languages, application generators)
- The issues concerning constructor generalization are:
  - programming languages and their evolution,
  - software adaptation and reuse, and
  - the fundamental programming mechanisms: *composition*, *adaptation*, and *construction*.
- Generalizing the CAMP constructors would do nothing more than ease the burden of creating similar constructors, and has no benefit outside of the CAMP arena.
- Truly generalized constructors are implemented such that the fundamental programming mechanisms of the base programming language are enhanced.

Concerning the integration of a Generalized Constructor Capability within expert system development environments and reuse environments:

- We do not currently recommend the use of ART, or a similar expert system tool, for the development of the GCC. This decision must be postponed until we have developed a sufficient understanding of the expertise necessary to adapt reusable software components for specific applications.
- Extraction and adaptation tools will play a significant role in the usability of future reuse library systems, and in the successful reuse of components catalogued within the libraries.

- Object-oriented classification based adaptation on retrieval is the best integration of the GCC within reuse environments.
- Future reuse environments providing comprehensive adaptation support incorporate the essence of the GCC.

5.2 Summary of the Research Effort

While meeting the objectives of this STAS, we did not strictly adhere to the four tasks outlined in our approach to meet these objectives. Our primary objective was to investigate the feasibility of developing a *generalized* component construction capability that relieves some of the problems of domain-dependence and part maintenance, building on the concepts and experiences of the CAMP effort. Our secondary objective was to investigate the integration of component construction technology with existing and emerging software development environments.

The four tasks (see Section 1.2) were specified as follows: 1) analyze the CAMP methodology, 2) define the requirements for a generalized constructor capability, 3) investigate development environment integration, and 4) demonstrate the feasibility of the generalized constructor capability. Towards the end of Task 1, we began to investigate alternative generalized constructor approaches, in effect opening a Pandora's box. Although thought to be a relatively straightforward exercise, we were faced with the realization that selecting an appropriate alternative approach was infeasible.

There are several reasons why we were unable to select an alternative generalized approach, all of which stem from unexpected research findings. First, we learned that the CAMP constructors were hard-coded. This was a real surprise. It was generally assumed that there was a knowledge-based component schema representation and rule-based component construction foundation upon which we would be able to refine and build upon. Secondly, it was assumed that the selected approach would be a matter of choosing an appropriate constructor-constructor type strategy to solve the maintainability problem, and incorporating and encapsulating domain knowledge where appropriate to relieve the domain-dependence problem. Once we began to explore alternative approaches outside of the CAMP arena, we gained a new perspective of constructors. We learned of the central importance of adaptation (and how little we know about it), the implications of programming languages and their evolution, and the necessity of a domain analysis for the development of a domain-specific adaptation and reuse capability. As a result of all of this, it was not possible to proceed with Task 2. However, our newfound insight provided us with a better understanding of constructors and their generalization.

We then proceeded to Task 3, the investigation of development environment integration. The expert system investigation led us to the conclusion that we have yet to develop a sufficient understanding of the expertise necessary to adapt reusable software components for specific applications. If, after developing the GCC requirements and/or a GCC specification, it becomes clear that some aspect(s) of the GCC are best suited for an expert system, then it would be appropriate to impose expert system implementation requirement(s).

The reuse environment investigation turned out to be surprisingly interesting and educational. Our previous work (from Task 1) had taught us of the synergy between the GCC, adaptation, and reuse. During Task 3, we built upon this observation, learning that not only was there a good integration between the GCC and reuse environments, but that reuse environments have a much needed automated adaptation capability. This investigation grew to include a detailed operational scenario and a significant amount of analysis. Thus, while we have not addressed Task 4 in the manner originally planned, we believe that this effort (Task 3) serves as a demonstration of the GCC's feasibility.

As far as defining the GCC requirements (as called for in Task 2), we believe that the scope of the requirements definition was larger than anticipated, and additional research is required. We have observed that there is an enormous amount of basic research in progress which is potentially applicable to the automated adaptation and reuse of software components. While promising, none yet have demonstrated the capability to scale-up to the demands of software engineering in-the-large.

A single organization such as CECOM cannot possibly conduct such a suite of basic research programs. Instead, those efforts which seem most promising need to be identified for further research and development. In addition, we believe that no single technique will be appropriate and applicable to the entire Command and Control (C^2) domain, or any other broad domain. The appropriate techniques need to be applied to the various subdomains, subsystems, and components of the C^2 domain. This can only be done by analyzing the intended target applications, their commonality, and their adaptation requirements. In addition to the continued basic research which is applicable to Computer Science in general, we believe that any domain-specific strategy needs to include a thorough investigation of the domain in question.

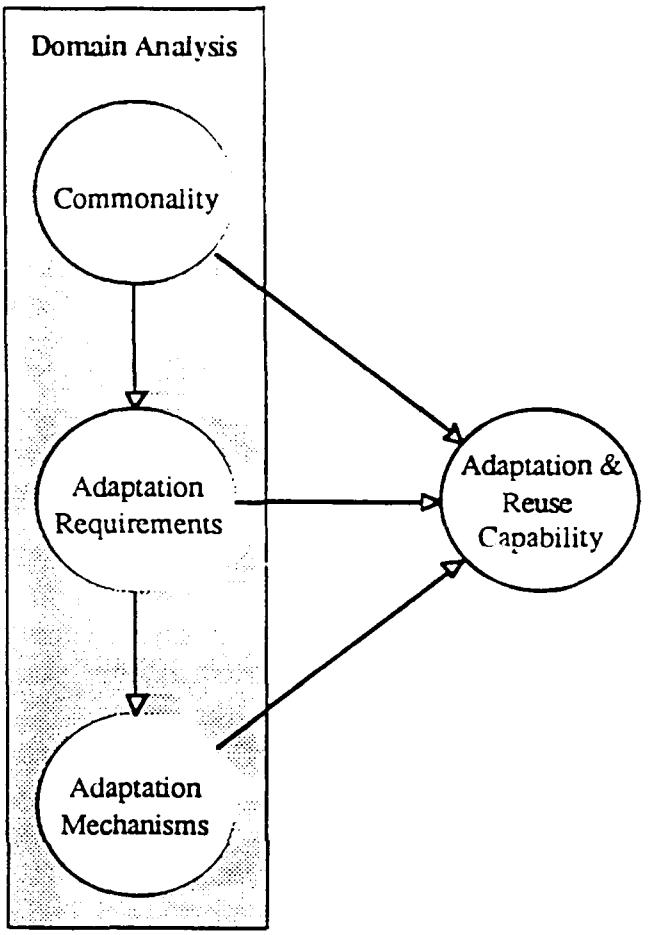


Figure 5–1 A Domain-Specific Adaptation and Reuse Strategy

Figure 5–1 illustrates the necessary steps which need to be taken in order to develop an adaptation and reuse capability specific to a large domain such as C^2. Domain analysis has been identified as a key requirement. To insure that the resulting components are adaptable, the domain analysis must include a strong level of adaptation awareness. Commonality studies will identify components, subsystems, and architectures which are candidates for development and reuse. In conjunction with this, the careful specification of adaptation requirements, followed by the selection and implementation of the appropriate adaptation mechanisms, will insure the development of highly adaptable and easily reused components.

Only this high level of adaptability, accompanied with the appropriate automation, will allow components to be successfully reused. A careful domain analysis is the only means by which to identify the necessary components (of varying scope and complexity) and the associated automation required to achieve sufficient adaptability. We believe that *a domain analysis is essential for the development of a domain-specific adaptation and reuse*

capability. It is this adaptation and its automation which is the Generalized Constructor Capability.

5.3 Recommended Future Directions

At the onset of this STAS, it was assumed that we would be able to pick a desired approach from a small number of alternative component construction approaches. This of course would be quite naturally followed by the requirements specification of one or more corresponding reuse tools. Unexpectedly, this strategy proved to be infeasible. In this section, we propose a new strategy for developing the GCC. The new approach includes a significant amount of research and detailed study. We believe this level of effort is necessary to successfully derive the GCC requirements. This section will detail our approach and provide specific recommendations.

We strongly recommend that CECOM continue to fund the research and development of a GCC. We consider the current effort to have been a necessary but preliminary effort. With the preliminary research complete, we are now ready to embark on a journey towards the specification of the GCC requirements. Our approach is illustrated in Figure 5–2. The three primary elements of the approach are a C^2 domain study, a languages/systems study, and basic research on software adaptation. The adaptation research is the central effort, overlapping both of the other studies. The domain study is a top-down analysis, investigating the larger aspects of the C^2 domain (e.g., generic architectures, subsystems). The languages/systems study is a bottom-up analysis, investigating fundamental programming mechanisms, constructs, and techniques.

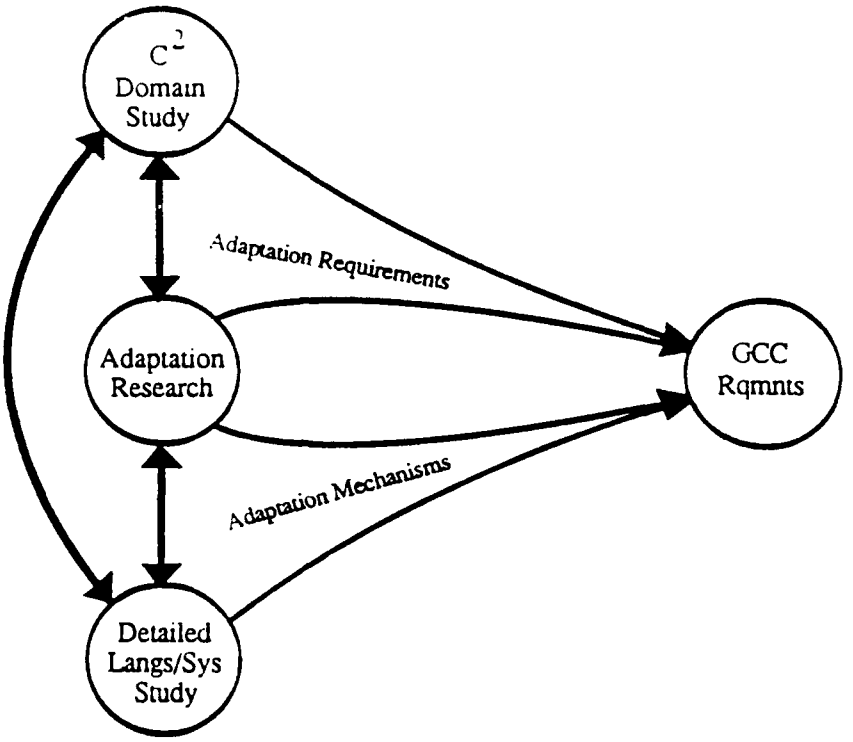


Figure 5–2 Developing C^2 GCC Requirements

The C^2 **domain study** is a scaled-down version of a domain analysis. A full domain analysis is not required to develop an initial GCC. As illustrated in Figure 5–1, the domain study should identify commonalities, adaptation requirements, and adaptation mechanisms. An emphasis should be placed on specifying the appropriate adaptation requirements. At this point, only limited effort should be exerted on the specification of corresponding adaptation mechanisms. The domain study should not proceed into the development (implementation) of the components or the adaptation mechanisms. The domain study is primarily a specification effort. This will be sufficient for the purposes of developing the initial GCC requirements.

The **languages/systems study** is an in-depth and focused continuation of the current research effort. Most of the current effort has focused on programming language issues, and specific R&D efforts and their prototypical/operational systems. This preliminary investigation was broad and shallow, and should be followed up with a detailed study of selected topics. This study should focus on potential enhancements to the Ada programming language and support environment. Constructs from other programming languages and concepts/techniques from R&D systems should be identified, analyzed, and evaluated.

The adaptation research is a broad yet focused basic research effort. One of the most important results of the current effort has been the identification of software adaptation as a central concern to software reuse. In order to benefit from this realization, an in-depth research effort should be launched on this topic. Our experience has taught us that this is a difficult topic to analyze, and it is especially difficult to reason about software adaptation in isolation. Adaptation is at the heart of both the domain study and the languages/systems study. The three studies, conducted in parallel, will support each other through overlapping boundaries.

The results of these studies should then be compiled and synthesized into the GCC requirements (see Figure 5–2). The combination of the domain study and the adaptation research should provide the necessary commonality and adaptation requirements. The combination of the languages/systems study and the adaptation research should provide the appropriate adaptation mechanisms (and their implementation) for each specified adaptation requirement. In addition, the languages/systems study should identify specific programming language constructs and techniques which may be appropriate for C^2. From all of this, the GCC requirements should then be synthesized through the assignment of specific adaptation requirements and mechanisms to the appropriate components, subdomains, and/or generic architectures. In addition, GCC requirements specifying programming language enhancements, and/or specific tools to support the GCC should be derived.

Clearly, it would be unwise to place restrictions and/or stipulations on the format, structure or contents of the GCC requirements at this time. The next phase of this R&D effort should be allowed the flexibility of responding to the front-end investigative research which will be conducted prior to the synthesis of the GCC requirements.

We believe that adopting and supporting the proposed approach will result in the successful specification of the C^2 GCC requirements. While the C^2-specifics are yet unknown, we believe that a GCC is not only feasible, but has great potential. A well-engineered GCC would undoubtedly have a significant impact on software reuse, resulting in the increased productivity, quality, and reliability of Command and Control software development efforts.

Our specific recommendations for future R&D are presented in the following sections. It should be noted that CECOM is already in the position to take advantage of several ongoing research efforts which together cover our recommendations:

- Adaptation Research/ARCS — Checkout Tools have been conceptualized and may serve as a solid foundation for GCC

implementation. Protocols, scenarios, and preliminary adaptation requirements have been defined. As the program develops, knowledge about object-oriented classification of reusable software components and their adaptation will be gained.

- Domain Study/SEI C^3I Domain Analysis — The level of intensity of the domain study does not need to be intense. The SEI domain analysis should provide a significant amount of information from which to develop an initial understanding of the domain-specific GCC.
- Programming Languages Study/Ada 9X — The Ada 9X project is currently evaluating numerous language enhancements to the Ada programming language. Eventually, the 9X version of Ada will contain many new features/constructs/enhancements. It is possible that the 9X version may have a significant impact on adaptation, generic architectures, and reusability in general.

5.3.1 Software Adaptation Research

A basic research effort should be conducted to explore software adaptation. In particular, the following aspects of software adaptation should be specifically investigated:

- Adaptation requirements and mechanisms
- The adaptation of software components of varying size and complexity (e.g., subsystems, generic architectures)

A study of adaptation requirements and mechanisms should minimally include identification and classification tasks. Other relevant topics which could also be investigated include multiple mechanisms (for a single requirement) and mechanism implementations.

The adaptation of complex components and/or groups of components is of special concern, especially considering the domain in question. For example, Quanrud describes a generic architecture as "an adaptable application." [QUA88] A special investigation should be made to explore this type of adaptation and how it can be supported (e.g., by object-oriented inheritance and specialization).

5.3.2 C^2 Domain Study

A Command and Control (C^2) domain study is essential to the specification of the C^2 GCC requirements, and to the eventual development of a GCC. The level of effectiveness of the resulting GCC will depend upon the scope and depth of the domain study. Specifically, the domain study should include the following:

- The identification of reusable C^2 components, subsystems, and generic architectures
- The specification of adaptation requirements and mechanisms which are best suited for the reusable C^2 components

As the domain study is being performed, and especially after it has been completed, the resulting work products should be analyzed for C^2-specific reuse opportunities (e.g., potential C^2 domain or subdomain languages).

5.3.3 Programming Languages/Systems Study

A detailed study of selected programming languages and R&D systems should provide the technology to support an advanced adaptation and reuse capability. Specifically, the following investigations should be performed:

- Consider adapting constructs from other programming languages and incorporating them into Ada
- Conduct a detailed study of promising R&D and/or commercial systems (e.g., Unisys' SSAGS, AT&T's Stage [CLE88], Classic-Ada)

Both of these studies must begin with a process of identification. Surveys may be necessary to aid this process. Once the candidates have been identified, they will be studied in detail. Candidates should be evaluated with respect to their applicability to the C^2 domain.

Close attention should be paid to the Ada 9X project. For example, support for subprogram types is currently under 9X review. The suite of candidate and/or approved Ada language issues needs to be monitored and taken into consideration.

Finally, the object-oriented paradigm should receive particular attention. As advocated by Quanrud, a key advantage of inheritance is that it provides adaptation and extension capabilities "without changing the code of the original reusable component and without anticipating the need to make any of those changes." [QUA88] The object-oriented paradigm thus supports reuse in a unique and very powerful way: it facilitates unforeseen adaptation.

References

[BAL82] Balzer, Robert M., *Operational Specification as the Basis for Rapid Prototyping*, ACM SIGSOFT Software Engineering Notes, December 1982, pp. 3–16.

[BAL85] Balzer, Robert M., *A 15 Year Perspective on Automatic Programming*, IEEE Transactions on Software Engineering, November 1985, pp. 1257–1268.

[BAL87] Balzer, Robert M., *Living in the Next-Generation Operating System*, IEEE Software, November 1987, pp. 77–85.

[BAS87] Bassett, Paul G., *Frame-Based Software Engineering*, IEEE Software, July 1987, pp. 9–16.

[BIG87] Richter, C. and T. Biggerstaff, *Reusability Framework, Assessment, and Directions*, IEEE Software, March 1987, pp. 41–49.

[CLE88] Cleaveland, J. C., *Building Application Generators*, IEEE Software, Vol. 5, No. 4, July 1988, pp. 25–33.

[EGE89] Ege, Raimund K., *Direct Manipulation User Interfaces Based on Constraints*, Proceedings of the 13th Annual International Computer Software & Applications Conference, IEEE Computer Society Press, September 1989, pp. 374–380.

[FRE87] Freeman, Peter, *A Conceptual Analysis of the Draco Approach to Constructing Software Systems*, IEEE Transactions on Software Engineering, July 1987, pp. 830–844.

[GEV87] Gevarter, William B., *The Nature and Evaluation of Commercial Expert System Building Tools*, Computer, Vol. 20, No. 5, May 1987, pp. 24–41.

[KAI87] Kaiser, Gail E. and David Garlan, *Melding Software Systems from Reusable Building Blocks*, IEEE Software, July 1987, pp. 17–24.

[KER84] Kernighan, Brian W. and Rob Pike, *The UNIX Programming Environment*, Prentice-Hall, Inc., 1984.

[KOV89] Kovarik, Vincent J. Jr., *Knowledge Base Assembler*, Software Productivity Solutions, Inc., SBIR Phase I Final Report, Contract No. DAAA21–88–C–0142, U.S. Army AMCCOM, March 1989.

[LELSS] Leler, Wm, *Constraint Programming Languages, Their Specification and Generation*, Addison-Wesley Publishing Company, 1988.

[LUQ88] Luqi, *Knowledge-Based Support for Rapid Software Prototyping*, IEEE Expert, Winter 1988, pp. 9–18.

[MCD85a] McDonnell Douglas Astronautics Company, *Software Requirements Specification for the Ada Missile Parts Engineering Expert System of the Common Ada Missile Packages (CAMP) Project*, September 1985.

[MCD85b] McDonnell Douglas Astronautics Company, *Software Top Level Design Document for the Ada Missile Parts Engineering Expert System of the Common Ada Missile Packages (CAMP) Project*, September 1985.

[MCD86] McDonnell Douglas Astronautics Company, *Software Detailed Design Document for the Ada Missile Parts Engineering Expert System of the Common Ada Missile Packages (CAMP) Project (Draft)*, August 1986.

[MCD87a] McDonnell Douglas Astronautics Company, *User's Guide for the Missile Software Parts of the Common Ada Missile Packages (CAMP) Project*, October 1987.

[MCD87b] McDonnell Douglas Astronautics Company, *Software Users Manual for the Ada Missile Parts Engineering Expert System of the Common Ada Missile Packages (CAMP) Project*, November 1987.

[MCD89] McDonnell Douglas Missile Systems Company, *Software User's Manual for the Common Ada Missile Packages – Phase 3 (CAMP–3) Parts Engineering System Catalog*, November 1989.

[MCF85] McFarland, Clay and Terry Rawlings. *DARTS: A Software Manufacturing Technology*, General Dynamics Data Systems Division, Western Center, San Diego, California. Presented at the STARS Application Workshop, April 1985.

[MCN86a] McNicholl, D., et. al., *Common Ada Missile Packages (CAMP) Volume I: Overview and Commonality Study Results*, McDonnell Douglas Astronautics Company, May 1986.

[MCN86b] McNicholl, D., et. al., *Common Ada Missile Packages (CAMP) Volume II: Software Parts Composition Study Results*, McDonnell Douglas Astronautics Company, May 1986.

[MCN86c] McNicholl, D., et. al., *Common Ada Missile Packages (CAMP) Volume III: Part Rationales*, McDonnell Douglas Astronautics Company, May 1986.

[MCN88a] McNicholl, D., et. al., *Common Ada Missile Packages — Phase 2 (CAMP-2) Volume I: CAMP Parts and Parts Composition System*, McDonnell Douglas Astronautics Company, November 1988.

[MCN88b] McNicholl, D., et. al., *Common Ada Missile Packages — Phase 2 (CAMP-2) Volume II: 11th Missile Demonstration*, McDonnell Douglas Astronautics Company, November 1988.

[MCN88c] McNicholl, D., et. al., *Common Ada Missile Packages — Phase 2 (CAMP-2) Volume III: CAMP Armonics Benchmarks*, McDonnell Douglas Astronautics Company, November 1988.

[MEY87] Meyer, Bertrand, *Reusability: The Case for Object-Oriented Design*, IEEE Software, March 1987, pp. 50–64.

[NEI84] Neighbors, James M., *The Draco Approach to Constructing Software from Reusable Components*, IEEE Transactions on Software Engineering, September 1984, pp. 564–574.

[QUA88] Quanrud, Richard B., CECOM Center for Software Engineering, "Generic Architecture Study," C04–038NN–0001–00, Final Report delivered by SofTech, Inc., January 22, 1988.

[RIC88a] Rich, Charles and Richard C. Waters, *Automatic Programming: Myths and Prospects*, IEEE Computer, August 1988, pp. 40–51.

[RIC88b] Rich, Charles and Richard C. Waters, *The Programmer's Apprentice: A Research Overview*, IEEE Computer, November 1988, pp. 11–25.

[SIM88] Simos, Mark A., *The Domain-Oriented Software Life Cycle: Towards an Extended Process Model For Reusability*, Tutorial: Software Reuse: Emerging Technology, edited by Will Tracz, IEEE Computer Society Press, 1988, pp. 354–363.

[SOW] Statement of Work for *Reuse Tools to Support Ada Instantiation Construction*.

[VOL85] Volpano, Dennis M. and Richard Kieburtz, *Software Templates*, Proceedings of the 8th International Conference on Software Engineering, IEEE Computer Society Press, August 1985, pp. 55–60.

[WAT85] Waters, Richard C., *The Programmer's Apprentice: A Session with KBEmacs*, IEEE Transactions on Software Engineering, November 1985, pp. 1296–1320.

[WAT86] Waters, Richard C., *KBEmacs: Where's the AI?*, The AI Magazine, Spring 1986, pp. 47–56.

[WIN81] Winston, Patrick H. and B. Horn, *LISP*, Addison-Wesley Publishing Company, 1981.

Appendix A — Glossary of Acronyms

ADT	Abstract Data Type
AI	Artificial Intelligence
AMPEE	Automated Missile Parts Engineering Expert [System] (CAMP)
ARCS	Automated Reusable Component System (SPS)
ART	Automated Reasoning Tool (Inference Corporation)
ASL	Application-Specific Language (Unisys)
BNF	Backus-Naur Form
C^2	Command and Control
CAMP	Common Ada Missile Packages
CASE	Computer Aided Software Engineering
CECOM	Communications-Electronics Command
COBOL	Common Business Oriented Language
COTS	Commercial Off-the-Shelf
CSC	Computer Sciences Corporation
CT	Checkout Tool (ARCS Reuse Library System)
DARTS	Development Arts for Real-Time Software (General Dynamics)
DIP	Dual In-line Package
FSM	Finite State Machine
FTR	Final Technical Report
GCC	Generalized Constructor Capability
HLL	High-Level Language
HOL	Higher Order Language
ISI	Information Sciences Institute (USC)
KBEmacs	Knowledge-Based editor in Emacs (Programmer's Apprentice)
LISP	List Processing (programming language)
LLCSC	Lower Level Computer Software Component
MIS	Management Information System
MIT	Massachusettes Institute of Technology
PA	Programmer's Apprentice (MIT)
PCS	Parts Composition System (CAMP)
PES	Parts Engineering System (CAMP)
PL/I	Programming Language/One

PSDL	Prototype System Description Language (Naval Postgrad. School)
RLS	Reuse Library System (ARCS)
RTE	Real-Time Embedded
SBIR	Small Business Innovation Research
SOW	Statement of Work
SPS	Software Productivity Solutions, Inc.
SSAGS	Syntax and Semantics Analysis and Generation System (Unisys)
STARS	Software Technology for Adaptable, Reliable Systems
STAS	Short Term Analysis Service
TIM	Technical Interchange Meeting
TLCSC	Top-Level Computer Software Component
UIMS	User Interface Managament System
VHLL	Very High-Level Language (4th generation)